Formal Analysis of Security Protocols Based on Web Services

Fatima Shabbir

August 8, 2011

UMI Number: U585480

All rights reserved

INFORMATION TO ALL USERS
The quality of this reproduction is dependent upon the quality of the copy submitted.

In the unlikely event that the author did not send a complete manuscript
and there are missing pages, these will be noted. Also, if material had to be removed,
a note will indicate the deletion.

UMI U585480
Published by ProQuest LLC 2013. Copyright in the Dissertation held by the Author.
Microform Edition © ProQuest LLC.
All rights reserved. This work is protected against
unauthorized copying under Title 17, United States Code.

ProQuest LLC
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106-1346

Contents

1 Introduction **1**

1.1 Web Services 2

1.1.1 Web Services Security 3

1.1.2 The Need for New Security Specifications for Web Services 6

1.1.3 XML-based Web Services Security Specifications 7

1.1.4 WS-* based Web Services Security Specifications . . . 10

1.1.5 Security Requirements 15

1.1.6 Vulnerabilities of WS-* Specifications 16

1.2 Formal Methods 19

1.2.1 Model Checking 22

1.2.2 Spin Model Checker 23

1.3 Problem Formulation 24

1.3.1 Research Objectives 24

1.3.2 Proposed Analysis 25

1.4 Attack Model 27

1.5 Thesis Overview 29

2 Literature Review **31**

3 System Model **49**

3.1 Chapter Objectives 49

3.2 Model Checking 50

3.2.1 Model Checking Process 50

3.3 Modelling Cryptographic Protocols 51

3.3.1 Guidelines for Modelling Cryptographic Protocols . . . 53
3.4 Security Protocol Models 56
3.4.1 Simple Message Exchange Protocol 57
3.4.2 Security Token Protocol 61
3.5 Environment Model for Protocols 65
3.5.1 Environment Model for SMEP 67
3.5.2 Environment Model for STP 71
3.6 Intruder Model for Protocols 76
3.6.1 Manipulated Protocol Run for SMEP 77
3.6.2 Manipulated Protocol Run for STP 78
3.7 System Properties 79
3.7.1 Property Specification for SMEP 80
3.7.2 Property Specification for STP 82
3.8 Concluding Remarks 84

4 Modelling Protocols with Automata **87**
4.1 Chapter Objectives 87
4.2 Pushdown automaton 88
4.3 Simple Message Exchange Protocol 90
4.4 Security Token Protocol 102
4.5 Concluding Remarks 116

5 Promela Model **118**
5.1 Chapter Objectives 119
5.2 Introduction to Promela 121
5.3 System Modelling Steps 122
5.3.1 Simple Message Exchange Protocol 125
5.3.2 Security Token Protocol 139
5.4 Concluding Remarks 155

6 Simulation and Verification Results **156**
6.1 Chapter Objectives 157
6.2 Simulation Results 158
6.2.1 Simple Message Exchange Protocol 158
6.2.2 Security Token Protocol 167
6.3 Verification 174
6.3.1 Simple Message Exchange Protocol 177
6.3.2 Security Token Protocol 178
6.4 Concluding Remarks 179

7 XML Injection Attack Model **181**
7.1 Simple Message Exchange Protocol 183
7.1.1 Types 183
7.1.2 Channels 184
7.1.3 Global Variables 185
7.1.4 Principal Processes 185
7.2 Security Token Protocol 191
7.2.1 Types 192
7.2.2 Channels 193
7.2.3 Global Variables 193
7.2.4 Principal Processes 194
7.3 Concluding Remarks 201

8 Simulation and Verification for XML Injection Attack **202**
8.1 Simulation Results 202
8.1.1 Simple Message Exchange Protocol 202
8.1.2 Security Token Protocol 208
8.2 Verification Results 216
8.2.1 Simple Message Exchange Protocol 216
8.2.2 Security Token Protocol 217
8.3 Concluding Remarks 217

9 Conclusions and Future Work **220**

Bibliography **229**

Appendix: SMEP **242**

Appendix: STP **245**

List of Figures

1.1 Web Services Security Stack. 14
1.2 SOAP envelope incorporating WS-Security 15
1.3 Message tampering scenario in a secure session 28
1.4 An XML rewriting attack 29

3.1 Schematic view of Model Checking approach 52
3.2 Security Token Protocol 66

4.1 Simple Message Exchange Protocol automaton 93
4.2 Security Token Protocol Protocol automaton. 107

5.1 Types of Objects 120

6.1 Message sequence chart for A → B 160
6.2 Message Sequence Chart for A→I 162
6.3 Message Sequence Chart for A → I followed by I(A) → B. . . 164
6.4 Message Sequence Chart for I → B. 165
6.5 Message Sequence Chart for I → B followed by A → I(B). . . 167
6.6 Message Sequence Chart for A → STS. 169
6.7 Message Sequence Chart for I → STS 171
6.8 Message Sequence Chart for I(A) → STS. 173

8.1 Message Sequence Chart for SMEP XML Content Injection Attack. 205
8.2 Message Sequence Chart - SMEP XML Element Injection Attack. 207

8.3 Message Sequence Chart for an STP Username Token Injection Attack. 211
8.4 Message Sequence Chart for an STP RST Content Injection Attack. 213
8.5 Message Sequence Chart for an STP RST Element Injection Attack. 215

9.1 Proposed architecture 228

List of Tables

4.1 Stacks for Simple Message Exchange Protocol 92
4.2 States and Transition Functions for SMEP. 100
4.3 Stacks for Security Token Protocol 106
4.4 States and Transition Functions for STP. 114

5.1 Promela Constructs 123
5.2 Simple Message Exchange Protocol Types. 126
5.3 Simple Message Exchange Protocol Global Variables. 130
5.4 Security Token Protocol - Types 140
5.5 Security Token Protocol Global Variables 146
5.6 Security Token Protocol Global Variables 147

6.1 Simulation results for SMEP. 168
6.2 Simulation Result for STP 174
6.3 Verification Results for SMEP. 177
6.4 Verification Results for STP. 178

7.1 Simple Message Exchange Protocol Types 184
7.2 Simple Message Exchange Protocol Global Variables. 186
7.3 Security Token Protocol Types 192
7.4 Security Token Protocol Global Variables. 195
7.5 Security Token Protocol Global Variables. 196

8.1 Simulation Result for SMEP 208
8.2 Simulation Result for Security Token Protocol 216
8.3 Verification Result for SMEP Under XML Injection Attack . . 216
8.4 Verification Result for STP Under XML Injection Attack . . . 218

DECLARATION

This work has not previously been accepted in substance for any degree and is not concurrently submitted in candidature for any degree.

Signed ..Fatima Shabbir.... (candidate)
Date ....10/Aug/2011...

STATEMENT 1
This thesis is being submitted in partial fulfillment of the requirements for the degree of PhD.

Signed .Fatima Shabbir... (candidate)
Date ....10/Aug/2011......

STATEMENT 2
This thesis is the result of my own work/investigations, except where otherwise stated. Other sources are acknowledged by explicit references.

Signed ..Fatima Shabbir. (candidate)
Date ....10/Aug/2011......
Signed ..Fatima Shabbir.
Date ....10/Aug/2011...... if accepted, to be available for photocopying and for inter-library loan, and for the title and summary to be made available to outside organizations.

Signed ..Fatima Shabbir...... (candidate)
Date ....10/Aug/2011......

ACKNOWLEDGEMENTS

I am thankful to my supervisors, Dr Coral Walker and Prof David Walker, whose encouragement, guidance and support throughout my PhD has enabled me to develop a deeper understanding of the subject. This thesis would not have been possible without their inspiration.

I also would like to thank my parents, Farrah and Shabbir Ahmed for all their encouragement. I would also like to thank Hasan, Omer and Areeb for their constant motivation and morale boosting.

Summary

This thesis examines the use of multi-stack pushdown automata to model the behaviour and properties of Web services based cryptographic protocols. The protocols are modelled in Promela and verified using the *Spin* model checker. The Simple Message Exchange Protocol and the Security Token Protocol are protocols that underlie the WS-Security and WS-Trust specifications, respectively. These two protocols are tested for correctness in the presence of an intruder that conforms to the Dolev-Yao model, i.e., it is tested whether the required properties the protocols hold in the presence of a Dolev-Yao intruder. The thesis also extends the Dolev-Yao intruder model to encompass attacks targeted specifically at Web services. An intruder model in Promela is created based on the Dolev-Yao abstraction which is extended to incorporate an XML injection attack model. The behaviour and properties of the Simple Message Exchange Protocol and the Security Token Protocol are then examined when subjected to an XML injection attack using this extended Dolev-Yao model.

CHAPTER 1

Introduction

Web Services are self-contained, modular applications that can be described, published, located and invoked over a network, generally the World Wide Web [Nei03].

The WS-* security specifications provide mechanisms for the security of SOAP messages, which are the most common way of communicating with, and between, Web services. The specification suite consists of various components each addressing different security requirements. For example, WS-Security, WS-Trust and WS-SecureConversation provide mechanisms to establish a shared security context to enable SOAP sessions. These specifications, however, are undergoing a process of maturation, and are therefore vulnerable to various threats. This work tries to address such threats through a formal analysis of these specifications.

1.1 Web Services

Rosenberg *et al.* [RR04] define Web services as follows:

> *"A Web service is an application that provides a Web API, identified by a Universal Resource Identifier (URI), whose interface and bindings are capable of being defined, described and discovered as XML artifacts. A web service supports direct interactions with other software agents using XML-based messages exchanged via Internet-based protocols."*

For Web services to be widely adopted they have to be made secure. The research community, together with industry, have introduced various specifications to make Web service platforms more secure. Different forms of these security specifications are available including WS-* based specifications and XML based security specifications. The WS-* base security stack includes *(i)* WS-Security, *(ii)* WS-Trust, *(iii)* WS-SecureConversation, *(iv)* WS-Policy, *(v)* WS-SecurityPolicy, *(vi)* WS-ReliableMessaging, *(vii)* WS-Federation, and *(viii)* WS-Reliability (see Section 1.1.4). XML based security specifications include *(i)* XML Signature, *(ii)* XML Encryption, *(iii)* XACML, *(iv)* SAML and *(v)* XKMS. Section 1.1.3 summarises these specifications.

These security specifications complement each other in order to improve overall Web service security. For example, the initial proposed WS-Security specification only addresses message level security and ignores various other aspects of security including trust establishment, reliable message delivery, and session level security. In order to address the establishment of trust, a

further extension, WS-Trust, was proposed.

Different categorizations of the security aspects for Web services have been introduced. For example, Nezhad *et al.* [Nea05] have categorized Web service security in terms of a five-dimensional space. These dimensions are: *(i)* secure messaging, *(ii)* resource protection, *(iii)* security properties binding, *(iv)* contractual interactions and *(v)* federated trust management. These dimensions help in evaluating any particular security specification. For example, the security requirements for messaging include confidentiality, integrity, non-repudiation, authentication and session security. Different security specifications address these security requirements. For example, confidentiality and integrity issues are both addressed by WS-Security. However these specifications are in their early development phase, thus leaving room for improvement. This work focuses on security involving WS-Security and WS-Trust.

1.1.1 Web Services Security

The sharing of global information, generated on different platforms by different applications, has emerged as an active research area over the past few years. Web services have developed as a globally accepted medium for sharing diverse information from different platforms. Web services are a transformational technology which is used to integrate resources inside and outside of an organization. The wider adoption of Web services is dependent on their security and reliability. Various efforts have been made by the research community to improve the security of Web services and as a result various security specifications have been introduced.

The following subsection presents some of the basic building blocks of

the Web services architecture. Following this, Sections 1.1.3 and 1.1.4 cover the two main families of security specifications currently available for making Web services platforms reliable and secure.

Messaging

Web Services employ SOAP messages for information exchange using the HTTP protocol. These SOAP messages are in XML format, as defined by the W3C SOAP Standard. SOAP defines many message exchange patterns (MEPs), for example one-way and peer-to-peer, but there are two main MEPs used in Web services: request and response. The general operation of Web services can be seen as request-response communication between two endpoints. When a certain SOAP request is received by a Web service, it returns a SOAP response after performing the corresponding operation.

Discovery Process

Web services use the Web Service Description Language (WSDL) and the Universal Description, Discovery and Integration (UDDI) service registry to support the service discovery process. WSDL is used to define the interface of a web service, whereas UDDI is used as a registry through which Web services can be discovered. Each Web service is responsible for creating a WSDL interface and may also enable dynamic binding. Using dynamic binding Web services can communicate with each other and with newly-added Web services. Web services can dynamically discover each other using UDDI.

Consider a service A looking for some operation O to be performed. Service B is capable of providing this service O and has its WSDL interface

registered with a UDDI registry. The following steps take place in order to discover service B:

1. Service B registers its WSDL interfaces with the UDDI registry so that its services can be discovered by other services.

2. Service A queries the UDDI registry for a service providing an operation O.

3. The UDDI registry returns the URI and details about accessing service B.

4. Service A utilizes B for the desired operation.

Portals

Web portals provide user-friendly interfaces for end-users, particularly in the context of a Service Oriented Architecture (SOA). Portals are platforms that are accessed through a Web browser, and which can be used as interfaces to a set of Web services. Instead of directly discovering the Web service, users can go to a portal and simply request the portal for a particular service. Some commonly-used web portals include Google, Amazon, and eBay. For example, in the Google portal, users request a particular piece of information and the portal finds the corresponding information (request) and returns the results (response) to the end user.

Roles

There can be different roles associated with a Web service. A Web service can act as *(i)* a provider, *(ii)* a requester, or *(iii)* an intermediary. The requester and provider roles are usually associated with endpoints. An intermediary

service is used to complete some overall task initiated by the requester. A provider Web service provides a response based on a request initiated by the requester Web service. It is also responsible for setting security requirements for authentication, authorization, encryption and non-repudiation. When a requester initiates a request, it can be its own request or on behalf of a another party. The requesting Web service is responsible for setting the proper syntax and security parameters required by the provider. Intermediary Web services are invoked as part of a chain of Web services, for example, XML Gateways. These roles are non-exclusive, that is, a service can be a requester at one instance and may be an intermediary at another instance.

Web Services Coordination

Web services can dynamically bind to each other in two different ways, *(i)* Orchestration and *(ii)* Choreography. Web service orchestration is performed within an organization. It allows the use of existing Web services to create another Web service. It has a central architecture, and Web services are invoked based on the decisions made by an orchestration engine. Web service choreography is performed between multiple organizations. Its invocation is more dynamic in nature and is based on relationships defined between individual services. Web service choreography is distributed in nature since there is no central control point.

1.1.2 The Need for New Security Specifications for Web Services

The traditional security mechanisms are not enough to secure Web services. Some of the reasons which give rise to the need to develop new security specifications for Web services include:

1. SSL/TLS based mechanisms only ensure the security of a message while the message is in the secure tunnel. The moment the message reaches an intermediary node, it is decrypted and becomes vulnerable to attacks (message contents become visible). SSL/TLS based security mechanisms only allow encryption/decryption of the whole message, whereas Web services may require the encryption of certain parts of messages. It may not be desirable in Web services to make the whole content of SOAP messages visible at intermediary points. Therefore, SSL/TLS based security mechanisms are not enough for securing Web services data.

2. When originally designed SOAP did not have any built-in mechanisms for security. It was only intended to provide interoperability between different platforms. The initial idea was that security should be added as an extension to SOAP. As a result different security specifications have been proposed to make SOAP messages secure.

3. A further complexity arises due to the fact that SOAP messages are communicated using the HTTP (or HTTPs) protocol. Traditional firewall filters may allow http traffic to pass through undetected. Both trusted and un-trusted users can initiate a Web service which passes undetected through the firewall.

1.1.3 XML-based Web Services Security Specifications

XML-based security specifications can be used for securing Web services [NKHBM04]. For example, XML encryption is used to achieve the confidentiality property for Web services. The following throws light on some of the existing XML-based security specifications used for Web services security.

XML Encryption

XML Encryption is a W3C security specification. XML data is text format data and thus additional encryption is required. SSL/TLS security mechanisms also use encryption, however, selective encryption cannot be done using SSL. XML Encryption on the other hand, allows selective encryption which is desirable for Web services to ensure confidentiality. In the context of Web services, XML Encryption [Nei03] allows the security principle of confidentiality to be satisfied across more than just the context of a single SOAP message. XML Encryption can also be used to keep the SOAP message encrypted even the message is processed by an intermediary Web service. XML Encryption involves expressing encrypted data using XML and allowing portions of the documents to be encrypted. Encryption can be performed on XML elements and contents, on only XML contents, and on arbitrary data.

Encryption steps include: *(i)* selection of an encryption algorithm, *(ii)* obtaining the encryption key, *(iii)* serialisation of data, *(iv)* performing encryption, and *(v)* specifying the data type. Decryption steps include: *(i)* determining the algorithm, parameters and ds:KeyInfo, *(ii)* locating the key, *(iii)* decrypting the data, *(iv)* processing the XML elements or XML element content, and *(v)* processing data that is not an XML element or XML element content.

XML Signature

XML Signature, introduced by W3C and IETF, facilitates the digital signing of XML data, and is used to provide data integrity in Web services [Nei03]. XML Signature allows three types of document signing: *(i)* the whole document can be signed, *(ii)* different parts of the document can be signed, and

(iii) different parties can sign the same document. XML Signature is also used for authentication and non-repudiation when used in conjunction with identity-based security. WS-Security employs XML Signature and binds the *"security token"* with the SOAP message. XML Signature may also be used for integrity and non-repudiation of WSDL files. An XML Signature may be used as: *(i)* an Enveloped Digital Signature, meaning that the signature is contained within the signed document, *(ii)* an Enveloping Digital Signature where the signed data is contained within the XML Signature structure itself, and *(iii)* a Detached Signature which is separate from the signed entities.

SAML

Security Assertion Markup Language (SAML) [Nei03] is an XML standard introduced by OASIS. It is used for exchanging authentication and authorization data between an identity provider and a service provider. OASIS SAML has three types of assertions: *(i)* authentication, *(ii)* attribute, and *(iii)* authorization. SAML is mainly concerned with access control for already authenticated principals based on some pre-defined policies. The SAML architecture has two roles: Policy Decision Point (PDP) for making decisions based on a set of policies, and Policy Enforcement Point (PEP) for enforcing the decisions. SAML also facilitates single sign-on for Web services.

XACML

XACML [Nei03] is used to control access to resources based on the characteristics of the requester, request protocol and authentication context in XML format. Access can be granted to resources by using *(i)* ACL (Access Control Lists) and *(ii)* RBAC (Role-Based Access Control). XACML is composed of different modules including *(i)* Policy Enforcement Point (PEP), *(ii)* Policy

Information Point (PIP), *(iii)* Policy Retrieval Point (PRP), *(iv)* Policy Decision Point (PDP), and *(v)* Policy Administration Point (PAP). For more details, readers are referred to [Nei03].

XKMS

XML Key Management Specification [Nei03] is service-oriented and, therefore, is implemented as a Web service. It supports management of public keys for Web services, and defines two Web services for this purpose: *(i)* the XML Key Registration Service specification manages functions related to life-cycle management of public key credentials, and *(ii)* the XML Key Information Service specification manages query operations that obtain and validate public key credentials.

The following table summarizes the XML based security specifications that are used for Web services.

XML Specification	**Usage**
XML Encryption	WS-Security, confidentiality
XML Signature	WS-Security, integrity, authentication and non-repudiation
SAML	WS-Trust, authentication and delegation
XACML	Authorization and privacy
XKMS	WS-Trust, trust establishment

1.1.4 WS-* based Web Services Security Specifications

The WS-* security specifications were introduced by a set of organisations including IBM, Microsoft, VeriSign, and many others. The WS-* family of specifications include various definitions that each address a specific security

issue in Web services. In the following, some of these security specifications are outlined, and their target applications are highlighted. we also summarise the possible vulnerabilities mentioned in these specification.

WS-Security

WS-Security was introduced as a security specification for securing Web services. It addresses the basic security requirements, and its further extensions address more specific requirements. The WS-Security specification [NKHBM04] defines means of securing SOAP messages by means of *security tokens* and *digital signatures* to allow secure message exchanges between applications. It also provides a means of associating security tokens with message contents.

WS-Trust

WS-Trust was introduced in order to address trust related issues for Web services security. WS-Security only deals with securing the message contents and does not address trust issues. The WS-Trust specification [NGG+07] is an extension to WS-Security. It allows secure communication between two parties, and focuses on two main aspects: *(i)* the issuing, renewing and validating of security tokens, and *(ii)* ways to establish and manage trust relationships and how to assess them.

WS-SecureConversation

In order to address the problem of redundancy in the WS-Security and WS-Trust specifications, a session-level security specification, WS-SecureConversation,

was introduced [Aea05]. The main idea here is to establish a shared security context between two endpoints so that the reestablishment of security credentials for each message can be avoided. It defines how to provide secure communication when multiple messages are exchanged. It provides a means of establishing and sharing a security context and deriving keys from them.

WS-Federation

WS-Federation aims at providing a security solution not only between two endpoints, but also between two organisations. The WS-Federation specification [Lea06] defines mechanisms through which different security realms can federate, i.e., control authorized access to resources in one realm to principals whose identities and attributes are managed in other realm. It defines mechanisms for the brokering of identity, attribute, authentication and authorisation assertions between realms, and privacy of federated claims.

WS-ReliableMessaging

The reliable messaging extension of the WS-Security specification suite ensures the reliable delivery of messages between two endpoints. It does not address issues specific to security, rather it aims at reliable delivery of messages. The WS-ReliableMesaging specification [Bea05] defines the reliable delivery of messages between a source and a destination, and a SOAP binding required for interoperability. It provides a mechanism to identify, track and manage the reliable delivery of messages.

WS-Policy

The WS-Policy specification [Bea06b] provides a general-purpose model and syntax for describing policies of a Web service. Security concerns involve tampering with policies and assertions, which should be signed.

WS-SecurityPolicy

WS-SecurityPolicy [DLea05] defines a framework for Web services to express their constraints and requirements, which are represented as policy assertions. This specification indicates the policy assertions for use with WS-Policy. It can be applied to WS-Security, WS-Trust and WS-SecureConversation.

The following table summarizes the WS-* based security specifications that are used for Web services.

WS-* Based Specification	Security Requirements Addressed
WS-Security	Confidentiality, integrity, non-repudiation, authentication
WS-Trust	Trust establishment, trust proxying
WS-SecureConversation	Confidentiality, integrity, non-repudiation, authentication and trust
WS-Federation	Trust federation
WS-ReliableMessaging	Security properties (availability)
WS-SecurityPolicy	Messaging security policies and trust policy

WS-Security is a communication protocol providing a means of applying security to Web services. The protocol works at the application layer and describes how to embed signatures and encryption headers in SOAP

messages. It does this by incorporating security features in the header of SOAP messages. WS-Security alone only ensures message level security. In order to meet other security requirements, such as the establishment of trust and session level security, there are various complementary specifications for WS-Security which, when used together, satisfy a broader range of security requirements. Figure 1.1 shows the WS-* stack.

Figure 1.1: Web Services Security Stack.

Figure 1.2 shows a SOAP envelope incorporating WS-Security, with unnecessary details omitted. The purpose of this example is to show how the WS-* standards are deployed at the SOAP level. As can be seen in this example, the security header is specified and has certain parameters which tell end systems the necessary information. The information given in green represents the security token, and that given in purple represents data relevant for a particular protocol run. For example, the “<nonce>” and “<created>” sub-elements are used to defend against replay attacks. A ‘nonce’ is a unique random value used once in a protocol run to help ensure that previous com-

munications cannot be re-used. In a replay attack valid data is transmitted/delayed with malicious intent. A replay attack can be carried out by either an initiator or an attacker in the middle of the protocol run.

```
<Envelope>
  <Header>
    <Security>
      ...
      <UsernameToken Id=2>
        <Username>USER-NAME</Username>
        <Nonce>cGxr8w2AnBUzuhLzDYDoVw==</Nonce>
        <Created>2008-08-04T16:49:45Z</Created>
        ...
      </UsernameToken>
      ...
    </Security>
  </Header>
<Body Id=1>... </Body>
</Envelope>
```

Figure 1.2: SOAP envelope incorporating WS-Security

In summary, some of the security challenges faced by Web services include repudiation of transactions, secure issuance of credentials, insecure services, the spread of viruses and trojan horses, denial-of-service attacks, incorrect service implementation, and the lack of quality of service due to improper design.

1.1.5 Security Requirements

Web services are popular because of their dynamic nature, platform independence, interoperability, and greater access to data. However, there are unresolved issues and problems, such as *(i)* data integrity and confidentiality of messages between end points and intermediaries, *(ii)* the integrity of the Web service in question has to be established beforehand, and *(iii)* the

availability of the Web service in case of attacks, such as denial-of-service. Perimeter based technologies are unfit to protect Web services because *(i)* they are dynamic and are not restricted or bound to one network, *(ii)* SOAP is transferred over HTTP which passes unhindered through firewalls, *(iii)* TLS is inadequate for protection of SOAP messages as it provides end-to-end security, but cannot accommodate Web services' inherent ability to forward messages to multiple Web services.

A combination of the security specifications discussed in Section 1.1.4 can be applied to make Web service use secure over the network. Some security techniques for Web services are as follows:

1. Web service confidentiality can be provided by means of XML Encryption.
2. The integrity of Web services can be ensured using XML Encryption.
3. Web service authentication and authorization can be performed by using SAML and XACML, as proposed by OASIS. Also, WS-Security can be employed for the purpose of confidentiality and integrity of SOAP messages, thus resulting in end-to-end SOAP message security,
4. PKI can be used for Web services using XKMS.

1.1.6 Vulnerabilities of WS-* Specifications

The WS-* specifications are in the process of maturation. WS-* specifications are not only vulnerable to threats specific to cryptographic protocols, but also to threats specifically targeted at them, such as WSDL Scanning, XML Injection, and so on. We summarise the security concerns

mentioned in the WS-* specifications[NKHBM04] [NGG+07] [Aea05][Lea06] [Bea05][Bea06b][DLea05] respectively as follows:

1. WS-Security. There are certain flaws in WS-Security which limits its deployment when used alone. As a result, further extensions have been proposed and WS-Security is used in conjunction with these extensions to provide a security solution. Some of the security concerns not addressed by WS-Security alone are: *(i)* freshness guarantee, *(ii)* protection of security tokens, *(iii)* certificate verification, *(iv)* using passwords without protection, *(v)* the use of randomness, *(vi)* man-in-the-middle attacks, and *(vii)* PKI attacks.

2. WS-Trust. One of the main security issues of WS-Trust is that the security token issuance messages are prone to tampering. To avoid this they should be signed. XML Signature can be used to sign the tokens. Security token requests are also prone to denial-of-service attacks. Symmetric keys and password-containing tokens should be sent to the concerned parties only. Tokens containing personal information should adhere to the security policy of the organisation. In multi-message exchanges, signatures are susceptible to attacks. To avoid this signature confirmation methods should be used.

   WS-Trust and WS-Security both work at the message level. The major drawback of this approach is that the trust and security processes have to be repeated for all the messages in a particular session between two endpoints. This leads to degradation of quality-of-service.

3. WS-SecureConversation. Some of the security considerations are: *(i)* replay attacks – to prevent replay attacks all relevant elements of a mes-

sage should be included in the signature, and security context establishment messages should be timestamped, *(ii)* security context establishment should contain all policies to prevent attacks like downgrading in which an attacker tries to downgrade encrypted message content to something that can be more easily exploited, such as clear text, and *(iii)* authenticating services are susceptible to denial-of-service attacks. This, however, is not a complete list of possible attacks.

4. WS-Federation. Common attacks to WS-Federation and their possible prevention include: *(i)* message alteration (include signatures of the message information using WS-Security), *(ii)* message disclosure (encrypt sensitive data using WS-Security), *(iii)* key integrity (use strongest algorithms possible, by comparing WS-Policy and WS-SecurityPolicy), *(iv)* authentication (WS-Security and WS-trust), *(v)* accountability (strong symmetric keys or PKI signatures), *(vi)* availability (one form of attack is replay and countermeasure is WS-Security), *(vii)* address spoofing (all addresses are signed) and replay (time-stamp mentioned in WS-Security), *(viii)* meta-data alteration (include signatures in meta-data or use secure channels for transfer), *(ix)* forged security tokens (Security Token Service must guard their keys to prevent forging of tokens and requester identities), *(x)* privacy (Security Token Service should not send requester's personal information without consent), and *(xi)* compromised services (if the Security Token Service is compromised it must not be able to issue tokens outside the compromised realm).

5. WS-ReliableMessaging. Common attacks to WS-ReliableMessaging and their possible preventions include: *(i)* message alteration (include

signatures of the message information using WS-Security), *(ii)* message disclosure (encrypt sensitive data using WS-Security), *(iii)* key integrity (use strongest algorithms possible), *(iv)* authentication (WS-Security and WS-Trust), *(iv* accountability (strong symmetric keys or PKI signatures), and *(v)* availability (one form of attack is replay and countermeasure is WS-Security).

6. WS-Policy and WS-SecurityPolicy. Security concerns involve tampering of policies and assertions, which should be signed. Unsigned policies should not be accepted.

1.2 Formal Methods

Our everyday lives are being governed more and more by computerised systems, ranging from small systems, such as mobile phones, to large systems, such as airplanes, industrial plants, and so forth. In critical systems, where investments or human life are involved, the quality of the system becomes of crucial importance. Such systems can be validated before deployment using formal verification techniques. The system, or part of it, can be modelled at an acceptable level of abstraction and checked for the properties that the system is supposed to possess. Some of the most famous software failures [BK08] which could have been avoided are the Intel Pentium II floating point bug, which caused a loss of about $475 million, and a defect in the software of the Mars Pathfinder spacecraft which had a disastrous effect. With proper verification, these errors could have been avoided.

Formal methods are a combination of mathematical and logical models of a system and its requirements [But01]. Formal methods have not

only been applied to software and hardware systems, but have also been extended to cryptographic protocols. Cryptographic protocols are aimed at providing security services across distributed systems. Some of the goals of cryptographic protocols include secrecy, authentication, integrity, and non-repudiation. The network is usually assumed to be hostile, in that it may contain intruders who can read, modify, and delete traffic, and who may have control of one or more network resources. As a result, security protocols are used in order to make the communication between two nodes secure. However, the difficulty in designing and analysing security protocols has been extensively debated over the past couple of decades. The factors responsible for complicating the analysis process include:

- The properties they are supposed to exhibit are extremely subtle.
- The environment of the communication network is hostile.
- Knowing the capabilities of intruders beforehand is extremely difficult.
- Security protocols are concurrent in nature which makes the analysis more challenging.

There has been a substantial amount of work done in the analysis of security protocols using formal methods. The need for this has arisen from the fact that so-called secure protocols have been proved not-so-secure at some later time. In order to increase trust in any security protocol the security requirements and promises need to be verified, and formal methods can be used in this regard. Some of the categories of formal methods are now briefly described.

Theorem Proving techniques correspond to a process where it is shown that some statement is a logical outcome of a set of statements. The problem is described in a logical form. Another technique is *Type Checking*, where the system is presented in the form of datatypes. Any difference in the types is considered as a threat to the system. Type checking is automatic and can handle infinite system states. The main drawback of type checking is that the type assertions have to be incorporated in the system at design time, making it a less scalable option.

In *Belief Logic* the possible states of the system are expressed as a set of rules, or "beliefs". These systems can be thought of as an "expert system" which has a knowledge base consisting of rules in the system. This concept was first proposed by Burrow, Abadi and Needham in their work on BAN logic [BAN90]. The main theme proposed in BAN logic is that the "beliefs" or "trust" in the system are presented as rules in the system. As in more recent forms of expert systems, new rules can be inferred from the existing rules. The verification process is then simply to look for any rule violations.

The work presented in this thesis is based on the *Model Checking* technique. The next two sub-sections describe the model checking process and give an introduction to the Spin model checking tool used in this thesis. Model checking is an automatic verification technique. The verification process is an exhaustive search of all the possible states in the system. It is fast and does not require mathematical proof. Model checking also generates counterexamples for the model under consideration. Logic can be expressed easily as temporal formulas. However, the main disadvantage of model checking is the state explosion problem.

1.2.1 Model Checking

Model checking based approaches can be described as a sequential process involving *(i)* modeling, *(ii)* specification, and *(iii)* verification. In the modelling phase we present the system in a formal notation having a finite set of discrete states. The specification phase deals with presenting the formal system in some form of mathematical or logical way, for example, temporal logic, predicates, finite state automata, and so on. The actual validation of the correctness of the model is done in the validation phase. The model checking systems are also referred to as state exploration systems, where the possible set of paths which an intruder may take are specified.

The model checking process consists of a system model describing the *behaviour* of the system. The system is represented as a finite state model and is automatically generated from a model description language, such as Promela, pi calculus, etc. This thesis presents a system model of the protocols used, and describes the environment these protocol will be active in. The properties of the protocol are given as formulas in Linear Temporal Logic. The protocol models are then translated into Promela and simulations and verifications are performed on them.

In some cases, the *system model* is automatically generated from a *model description* that is specified in some appropriate dialect of a programming language. These system models are accompanied by algorithms that systematically explore all states of the system. This leads to different verification techniques, such as model checking, simulation, or in testing in reality.

In order to describe *what* the system ought to do, property specification languages like Propositional Temporal Logic, or PTL, can be used. Such languages are used to express correctness properties of a system. They are extensions of propositional logic, and reason about the system in terms of time.

1.2.2 Spin Model Checker

This sub-section gives a brief description of the model checking tool used in our research. Spin (Simple Promela Interpreter) [Hol03] has been used for model checking various software systems. A verification model consists of a set of facts about the system which we want to verify, and aspects of the system which are needed to verify those facts. Spin can be used for verifying the correctness of verification models. These verification models can be described in a specification language. Spin is used in this thesis because it counters the state explosion problem by using the partial order reduction algorithms built into it, thus reducing the number of transitions and states in the system. Spin can generate reduced state space, with only representatives of classes of execution sequences that are indistinct for a given LTL property. Partial order reduction [Hol03] works with commutative property of concurrently executed traditions that result in the same state when executed in different order.

The specification language that Spin takes as input is Promela. Spin can be used in two modes: simulation and verification. Simulation provides a representation of types of behaviour of the system model. Verification is per-

formed to prove some facts about a system. One way of representing these facts is in the form of Linear Temporal Logic, as is explained in more detail in Chapter 3. XSpin is a graphical interface to Spin that provides a visual environment for the simulation runs.

Promela is an acronym for Process Meta-Language. Promela is a specification language used for describing abstractions of the system design, and is not an implementation language. The focus of Promela is on modelling of process synchronization and coordination, and not on computation.

1.3 Problem Formulation

"Security protocols are three-line programs that people still manage to get wrong." (Roger Needham).

1.3.1 Research Objectives

The main research objectives of this thesis are as follows:

(i) *Can Web services based cryptographic protocols (WSBCPs) be modelled using automata to reflect the 'operations' and 'properties' the protocol is supposed to satisfy?*

(ii) *Can the Dolev-Yao model be extended to cater for WSBCP-specific attacks using traditional model checking techniques?*

(iii) *Can WSBCPs and their goals be accurately modelled and analysed using existing model checkers, and can attacks based on the Dolev-Yao model be detected?*

The Dolev-Yao model is a formal model introduced by Dolev and Yao to prove the properties of communicating protocols. The details of the Dolev-Yao model are discussed in Section 1.4. A pushdown automaton is a finite state automaton which uses a stack containing data to decide which transition to take, as discussed further in Section 4.2.

1.3.2 Proposed Analysis

This thesis proposes to use a multi-stack pushdown automaton model to map the 'functionality' and 'properties' of WSBCPs. Two stacks are used for the two legitimate services A and B participating in the protocol run. Each participating service is allocated its own stack. The number of stacks depends on the services involved in the protocol. The stack contains the operations that must be applied to the service before proceeding to the next state in the protocol run. We deviate slightly from the traditional definition of stack elements. Stack alphabets are defined as functions. The combination of automaton and the stack expounds the 'functionality' and 'properties' of the WSBCPs that the protocol is supposed to satisfy. At the end of a successful run, the stack is empty and the protocol 'goals' are satisfied. The WSBCPs are modelled using traditional cryptographic notations (symbolic cryptography) and present the protocol environment as a transition system. Promela is used to model the protocol and its environment. Spin is used to perform simulations and verify the models. A protocol run is said to be correct with respect to some property when at the end of the run the protocol has satisfied that property throughout the run. For example, if the property of the protocol is secrecy, then at the end of the run the message contents should have remained secret thus implying a 'correct' run of the protocol. This thesis examines two protocols: the Simple Message Exchange Proto-

col (SMEP), based on WS-Security, and the Security Token Protocol (STP), based on WS-Trust. SMEP is said to be correct when it satisfies secrecy and authentication goals. STP is said to be correct if at the end of a run services agrees on a full security context, while maintaining authentication and secrecy. Linear Temporal Logic is used to specify these properties of WSBCPs for verification using Spin.

This thesis also studies the application of the Dolev-Yao model to WS-BCPs, and extends the model to encompass attacks targeted at Web services. An intruder model in Promela is created for the Dolev-Yao abstraction and our proposed XML Injection attack model. SMEP and STP are subjected to these attacks. This thesis studies the suitability of applying traditional cryptographic model checking techniques to WSBCPs. The thesis models and analyses WSBCPs using an existing model checker (Spin), and detects attacks based on the Dolev-Yao model.

The focal point of the thesis is a methodology based on automaton theory for modelling WS-* cryptographic protocols. The approach will allow us to model the properties and the behaviour of the protocols in a singular model. This approach can also be extended to protocols that are not described using WS-* specifications e.g TCP/IP, Needham and Schroeder protocol[NS78].

We use push-down automata to to model WSBCPs as the combination of the input tape, automaton and the stack allows us to capture the behaviour and properties of the protocols.

A blocking protocol is one where the protocol is blocked till the previous

step in the protocol has been satisfied. PDA will cause the SMEP and STP to reflect blocking behaviour. The next step in the protocol run will not be executed till the previous step has reached an end.

SMEP and STP protocols are synchronous in nature. Synchronous services are characterized by the client invoking a service and then waiting for a response to the request. The sender service will wait for the response from the receiving service before the execution of the next protocol step. However, the security token service can be extended to emulate asynchronous behaviour. With asynchronous services, the client invokes the service but does not or cannot wait for the response. When communicating with multiple clients, STS will not require to complete the existing protocol run. STS can initiate multiple protocol runs simultaneously.

1.4 Attack Model

It is assumed that *'A'* and *'B'* are the two endpoints initiating a conversation after establishing a secure session using WS-Security, WS-Trust and WS-SecureConversation. There is a possibility that an attacker can tamper with the contents of individual messages on their way from the source to the destination. In the presence of such an intruder with certain privileges, the aim is to validate whether the security specifications under study behave in a secure manner. Such tampering may go undetected, that is, the attacker may disguise itself as if it was the trusted source. The attacker may tamper with the message in order to achieve denial-of-service (DoS), replay attacks and initiate its own session with service *'B'*. Figure 1.3 illustrates the con-

cept in more detail. In this figure Service A sends messages to Service B. An attacker intercepts the messages, tampers with them, and then forwards the modified messages to Service B. This kind of tampering may go undetected, resulting in denial-of-service, replay attacks, etc.

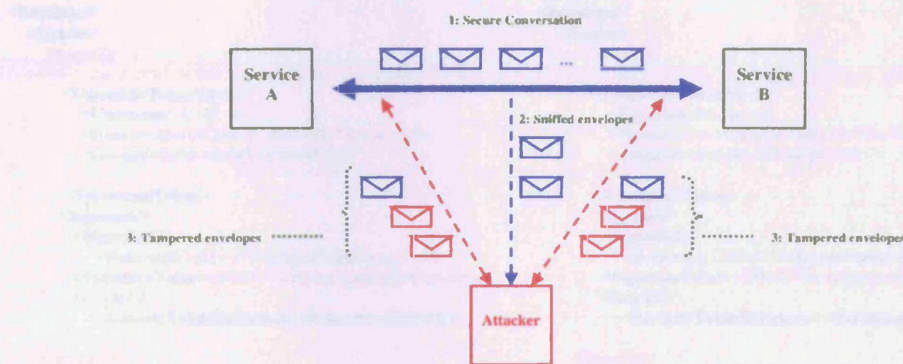


Figure 1.3: Message tampering scenario in a secure session

It is assumed that the attacker in our model has abilities as specified by the Dolev-Yao model. These are the ability to:

- Overhear and intercept all the messages over the network.
- Modify the messages.
- Generate new messages using information from overheard messages and some prior information.
- Send a new or captured message to another entity in the system.

It is also assumed that the underlying cryptography cannot be broken.

Figure 1.4 shows an XML rewriting attack. The sender A in this case is trying to send a fund transfer request to B. The message is intercepted

by the intruder, and modified so that the funds beneficiary is changed from A to 'Intruder'. This attack was only possible because the message body was not correctly signed. Although the password of the sender has not been broken, nevertheless, the attack was still possible because only the body was rewritten and the encryption key had already been trusted by the receiver.

```
<Envelope>
  <Header>
    <Security>
       ...
      <UsernameToken Id=2>
        <Username>A </>
        <Nonce>eGxr8w2AnBGzuhLzDYDoVw==</>
        <Created>2008-08-04T16:49:45Z</>
         ...
      </UsernameToken>
      <Signature>
        <SignedInfo>
          <Reference URI=#1><DigestValue>Ego0...</>
        <SignatureValue>vSB9JU/Wr8ykpAlaxCx2KdvjZcc=</>
        <KeyInfo>
          <SecurityTokenReference><Reference URI=#2/>
       ...
    </Security>
  </Header>
<Body Id=1>
    <TransferFunds>
    <beneficiary>B</>
    <amount>1000</>
 </Body>
</Envelope>
```

```
<Envelope>
  <Header>
    <Security>
       ...
      <UsernameToken Id=2>
        <Username>Alice </>
        <Nonce>eGxr8w2AnBGzuhLzDYDoVw==</>
        <Created>2008-08-04T16:49:45Z</>
         ...
      </UsernameToken>
      <Signature>
        <SignedInfo>
          <Reference URI=#1><DigestValue>Ego0...</>
        <SignatureValue>vSB9JU/Wr8ykpAlaxCx2KdvjZcc=</>
        <KeyInfo>
          <SecurityTokenReference><Reference URI=#2/>
       ...
    </Security>
  </Header>
<Body Id=1>
    <TransferFunds>
    <beneficiary>Intruder</>
    <amount>5000</>
 </Body>
</Envelope>
```

Figure 1.4: An XML rewriting attack

1.5 Thesis Overview

The rest of this thesis is organised as follows. Chapter 2 gives a review of the existing literature in the area. Chapter 3 defines the system model and gives the definitions for the WSBCPs used in this work. Chapter 4 contains the pushdown automaton model for the WSBCPs. Chapters 5 and 6 define the Promela models for the protocols and give simulation and verification results, respectively. Chapters 7 and 8 specify the XML injection attack

model, and give the simulation and verification results. Chapter 9 presents the concluding remarks, contributions, limitations, and future work.

CHAPTER 2

Literature Review

As the complexity of software and hardware systems increases over time, the adoption of formal methods for verifying the correctness of such systems becomes increasingly appealing. Software and hardware errors may not often threaten lives, but they sometimes have a serious financial impact – it is all about money [BK08]. Society is becoming increasingly dependant on computer networking, which in turn has lead to the adoption of cryptography in a variety of complex systems, e.g., financial transaction systems, online ticket reservation systems, Amazon, eBay, PayPal, etc. As systems grow more complex the threats to these system becomes manifold. Not only is it necessary to defend against intruders, but also against denial-of-service attacks and network traffic monitoring.

Over the course of the past two decades, researchers have been apply-

ing formal methods techniques to cryptographic protocols for verifying the correctness of the protocols. For the purpose of modelling and verification, not only have general-purpose tools been developed but also tools for specific tasks have evolved.

This chapter first presents some emerging trends and issues for cryptographic protocols and their analysis, and also presents research being done and gaps in this area. The current research directions and issues in the formal analysis of Web services based security protocols are then discussed. Cryptographic protocols use cryptography to distribute keys and data over a hostile network. A network is said to be hostile in the presence of an intruder who can read, modify and delete traffic.

In general, the types of formal methods used for analysing security protocols lie in following categories:

1. State exploration/model checking.
2. Theorem proving.
3. Type checking.

Model checking based approaches can be described as a sequential or an iterative process involving *(i)* modeling, *(ii)* specification and *(iii)* verification. In the modeling phase the system is presented in a formal notation in terms of a discrete finite set of states. The specification phase deals with presenting the formal system in a mathematical or logical way, for example, with temporal logic, predicates, finite state automata, and so on. The actual validation of the correctness of a model is done in the validation phase. Model

of showing that the state space was sufficient to guarantee security.

Finite state modelling does not give proof of security for the entire set of possible states, however it allows unambiguous statements about the conditions under which the deductions hold and an effective procedure for checking them. Finite state models are useful for analysing the security of cryptographic protocols as they allow an analysis of possible paths of the intruder to be made, and also allows assumptions about the environment of the system to be defined.

Theorem proving techniques are correlated to a process that shows that some statement is a logical consequence of a set of other statements. The finite state condition is loosened here. The system needs a precise description of the problem written in some logical form. On the other hand, type checking aims to present the system elements in the form of data types. The type variations in the system are then considered as possible threats in the system. Similar to a model checking system, type checking is also fully automatic, and has an added advantage of handling infinite system states. However, the main drawback of type checking is that the type assertions have to be incorporated into the system at design time, making it a less scalable option. Relevant work on type checking can be found in [Aba99, GJ03].

Recent research trends have been in state exploration and theorem proving techniques based on Dolev and Yao's model. Lowe [Low96a] showed that a general-purpose model checker can be used to find attacks in the Needham-Schroeder public key protocol. Lowe used the FDR model checker to find man-in-the middle attacks, and was one of the first to use a general-purpose

model checker for analysis of cryptographic protocols. This led to the application of theorem provers [Pau98, DS97] and model checkers [DK99, MMS97] to the problem. Work was also done on the design of purpose-built model checkers [Hui99, MCJ97, SBP01]. Research is still being done to show that the finite search space for model checkers is sufficient under certain situations [SBP01, MCJ97, AL00, FA01, MS01].

Millen developed CAPSL [DM00], a common authentication protocol language. Thayer developed the Strand Space Model [FHG98], a graphical representation of the Dolev and Yao model.

Paulson used the Isabelle theorem prover for analysis of cryptographic protocols. The main problem with theorem provers was the lack of counterexamples generated [Mea03, MR00, Coh00, HS00].

Belief logic, on the other hand, is different from state exploration techniques, and expresses the possible states of the system as a set of rules, or "beliefs". Such a system could be thought of as an "expert system" which has a knowledge base consisting of rules in the system. This concept was first proposed by Burrow, Abadi and Needham in their work on BAN logic [BAN90]. The main theme proposed in BAN logic was that the "beliefs" or "trust" in the system are presented as rules in the system. As in more recent forms of expert systems, new rules can be inferred from the existing rules. The verification process is then simply to look for any rule violations. For example, if a key is believed to belong to a particular participant, say 'X', then any message coming from 'X' signed by the same key is considered trustworthy. Whereas, if another participant, say 'Y', tried to use the same

key for sending a message, then it is considered to be illogical, and hence is reported as rule violation.

State exploration techniques are stronger than BAN Logic [BAN90], as BAN logic works at a higher level of abstraction. With the improvement in state exploration techniques, interest in BAN logic subsided. State exploration techniques are focused on exploring the possible paths in the system which could be taken by an intruder. The number of states must be kept sufficiently small so that the analysis can be performed in a reasonable amount of time.

Belief logic systems make the big assumption that the rule-making process is free of errors. They are considered to be less effective than state exploration systems [Mea03]. Lowe [HG01, Low97, Low96b] modelled protocols in CSP and applied a model checker to test its behaviour. These methods can detect attacks quickly but keeping the state space small requires assumptions which simplify the model and results in loss of accuracy. Paulson's work [Pau97, Pau99] also used an inductive approach. The protocol was represented as an infinite number of statements as a set of traces. Due to the large number of inductive definitions, the resulting proofs were complicated, however, they presented a more rigorous analysis.

The most recent approach to cryptographic protocol analysis is type checking [Aba99, BCJS04]. Security problems are identified by assigning message channels types and identifying type violations. The main drawback of this approach is the consideration of security violations when writing the specification. However, in model checking security violations are represented

as temporal logic and are independent of protocol specification.

Cryptographic protocols operate in different types of environments such as IKE[HC98] and SET[RRM], and thus are required to be more adaptable and complex. Increasing the complexity makes it difficult to verify the protocol. With the increase in types of threats, the analysis of these protocols becomes more critical. There are a number of different tools available for verifying safety properties, such as secrecy and authentication, for cryptographic protocols. These tools are based on the attacker model presented by Dolev and Yao. Due to the security of the protocol being an undecidable problem [EG83, HT96, CDL+99], the results of the tool are unsuccessful at times.

Some of the research gaps in the analysis of traditional cryptographic protocols include unbounded protocols, where the number of data fields is not limited. As the environment and the intruder behaviour become more complex there is a need for research into new threat models for protocols. Dolev and Yao's intruder model needs to be extended to handle these new threats, such as denial-of-service attacks, or else new threat models need to be developed and embedded into the current analysis techniques. Current tools cannot be directly used for the detection of these types of attack. Another challenge faced by the research community is in formal analysis of Web services based cryptographic protocols, which are more complex in nature, and not only encounter challenges in common with traditional protocols, but also possess their own set of problems.

With the advent of Web services, there has been an increase in the chal-

lenges faced when designing security protocols. In Web services technologies, the WS-* stack has been used for ensuring Web services security. However, Web services based security protocols not only face the normal challenges of cryptographic protocols, but also additional ones. In order to make Web services available across different platforms, simple XML based standards are introduced. The main idea is to use simple text-based messages for communicating between different parties. The architecture of such systems requires that certain portions of the XML based SOAP envelopes be encrypted. This complicates the security process since the unencrypted portions of a SOAP message leave it open to threats. WS-Security make use of XML Encryption and XML Signature to ensure general SOAP message safety, since these are the main security mechanisms for authentication and secrecy [Rea02, ERS01]. However, there have been certain enhancements to the basic WS-Security specification. The focus here is on trust establishment and session level security which is provided by WS-Trust and WS-SecureConversation [NGG+07, Aea05]. Further, only the work done in the area of formal analysis of the standards under discussion is presented.

Before the related work on the formal analysis of Web services is presented, it is important to discuss the use of the Dolev-Yao model for analysing the WS-* stack. Backes and Gob at IBM Research [BG05] have presented their findings on the use of the Dolev-Yao model for Web services security. They present a list highlighting the points which could be improved in the Dolev-Yao model assumptions when used for formal analysis of Web services security protocols. A similar set of guidelines was proposed by Meadows [Mea03] which pointed out new trends in security protocols and the abilities of attackers who are assumed to have more powers now than those assumed

in the Dolev-Yao model. One such ability of an attacker is cryptanalysis. Cryptanalysis is the study of methods for analysing and decyphering encrypted information [Jou09]. Backes' work on the other hand gives a more focused discussion for Web services.

Most of the work done in formal analysis is based on work done by Dolev and Yao. Although the Dolev-Yao model has been widely adopted by the research community for modelling the environment for cryptographic protocols, in the context of WSBCP, with their complex behaviour and the possibility of new attacks, there is now a need to modify the Dolev-Yao intruder model. Thus, the model needs to be tailored to WSBCP's. We also believe that ignoring the low level details may result in missing necessary requirements. The Dolev-Yao model is suitable for a behavioural analysis of Web services based security protocols but for more detailed analysis the model needs to be extended.

Blanchet *et al.* [Bla02] have presented a performance analysis of different security protocols. Their approach is to present the system in terms of belief logic, where rules for system transitions are presented in the form of predicates. They given a computational analysis of a number of rules needed for different security protocols, and the amount of time required to validate the rules. They have analysed both authentication protocols, Needham-Schroeder and Woo-Lam [WL93, Low96b, Sch98, BAN90], and the protocols involving session keys. Their results show that their fully automatic approach was ablc to dctcct flaws in protocols with no falsc alarms. Howcver, the state space in their work is kept small, and the assumptions made in order to keep the model small results in less accuracy.

Tobarra *et al.* [TCCD07] used a model checking approach to do a formal analysis of the WS-SecureConversation standard. In their work, they have used a high level formal language, HLPSL, for specifying system requirements. The requirements are then analyzed using one of four different verifiers available in the AVISPA architecture [ABea05]. The capabilities of the attacker are in accordance with the Dolev-Yao model [DY81]. Their work has highlighted that when used with other complementary WS-standards, like WS-Addressing [Cea04] and WS-Security, WS-SecureConversation could eliminate certain threats including replay attacks and false password attacks.

Kleiner *et al.* modeled Web service security using CASPER and FDR [KR05]. After giving the model of Web service security, they present an extension of ϕ (a mapping function from SOAP message to CASPER input) for modeling WS-SecureConversation. As in Tobarra *et al.* [TCCD07], they also conclude that including essential elements from WS-Addressing helps to control replay attacks. They also show that careless use of the Web Services Enhancement protocol (WSE), which allows developers to build secure Web services based on the latest Web services protocol specifications, may lead to denial-of-service attacks. The authors have commented that the correct authentication of clients should be used rather than relying on the mechanism of the WSE suite to detect replay attacks. WSE considers any subsequent Request Security Token messages from the same client as replay attacks. However, the client may be trying to establish multiple sessions with the server. They have presented their solution to bind the Request Service Token message with the client using negotiation. Their contribution is a formal translation of WS-SecureConversation to a traditional cryptographic proto-

col. They have presented their model at a higher level of abstraction, thus excluding the detailed structure of tokens in WS-SecureConversation. This high level of abstraction may result in errors being undetected.

Gordon *et al.* in their early work introduced the concept of "spi calculus" for modelling SOAP security headers [GP02]. They present a theoretical model of web security abstractions. This work formed the basis of a later version of their work which specifically focused on the WS-* stack. In this work, they have presented the ground work that how object calculus can be used to present the primitives for creating and calling Web services. They gave their own implementation of the proposed model. In their later work [BCFG04], they have extended their approach and have presented the model in the their proposed language, called "TulaFale" [BF04].

Bhargavan *et al.* proposed TulaFale which is based on predicate calculus [BF04]. In this work, the authors have successfully applied the pi calculus based TulaFale model to represent WS-Security. The system is presented in the form of messages passed over either a public channel or a private channel. The start of a communication process is marked with a "begin" message and there is an "end" message at the end. The "begin" and "end" identifiers are used to ensure the *authentication* and *correlation* requirements. The messages are expressed in the form of predicates. The authors show that in certain cases when time stamps are not signed, there could be replay attacks. They have shown that their "TulaFale" model detects these variations in the original traffic. The main predicates are presented in two forms. Constructor methods are used to apply some function to the given data, such as encryption. Destructors on the other hand do the reverse and extract the

information from the given data, such as in the decryption process. A predicate is defined as a logical combination of these constructor and destructor methods. Messages are created using "mkMsg" predicates and checked using "isMsg" predicates. The normal TulaFale messages are sent over the public channel, however, the secret keys are sent over the private channel. The protocol run is then analyzed using the ProVerif analyzer tool [Bla01, Bla02]. This work sets the foundation for extending it further to analyse WS-Trust and WS-SecureConversation.

The work of Bhargavan *et al.* on secure sessions for Web services applies the same model to WS-Trust and WS-SecureCoversation [BCFG04]. The model of these protocols is represented as a TulaFale language script. Their model pointed out certain vulnerabilities in the standards under study and proposed corrections. In a similar work, Bhargavan *et al.* use the "F#" language to model the WS-Security protocol and use ProVerif for verification purposes [BFG06, BFGT06]. The work in this thesis is primarily inspired by the TulaFale work. However, we believe that the model can be presented in a more dynamic and flexible way by using multi-stack pushdown automata for specifying system functionality.

One limitation to the pi calculus approach in [BF04, BCFG04, BFG06, BFGT06] is the modelling of non-determinism. The non-determinism in the pi calculus approach is controlled by the attacker. Their model does not offer complete protection against replay attacks. A necessary addition to their work is that not only timestamps need to be signed, but also nonces should be signed for replay detection. Also, predicates satisfy properties by pattern matching and are sensitive to the structure of the message.

Decision procedures have been proposed for analysis of cryptographic protocols, and Chevalier [Cea07] applies these to Web services based security protocols. We believe the flexible format of the XML structure and partial parsing of the SOAP message makes it difficult for modelling and analysis. They have modelled services using deduction rules and equations that reflect all the possible operations of the participants. Their work is complimentary to the Samoa project [Res]. The main contribution of their work is development of a verification procedure that detects rewriting attacks.

Other prominent work on the formal analysis of Web services includes the Johnson model [JJLea04] for the Web service atomic transaction protocol in the TLA language and checked by the TLC model checker [LW09]. TLA fails to capture the WS-AtomicTransaction specification in detail. Backes [Bea06a] has conducted an analysis of WS-ReliableMessaging. Diaz *et al.* [DPC+06] have also approached this problem using model checking. Their work, however, focuses on time critical applications of Web services. They have discussed a realtime flight reservation system in which the transactions are time-bound. They have shown that the entire system state can be modelled in the form of finite state automata with an additional tagging of time along the edges. The transition decisions are constrained by the time elapsed from a given point. The concept of timeout, however, is already available in the WS-* stack and could be used to our advantage. Their work does not focus on the security aspects of Web services based security protocols.

Huang [HM06] has made a brief survey of model checking technologies suitable for Web services. Various models, such as OWL-S, BPEL4WS and

WSDL describe Web services, but are semi-formal. There is a need to generate formal models that are mathematically rigorous as required by a model checker. WSAT [Fea04] is a front-end tool for translating guarded automata into Promela accepted by Spin [Hol03]. UPAAL [Bea04] accepts realtime models and checks for timing constraints. BLAST [Hea02] is used for model checking workflows. Blade [TP05] implements proof slicing techniques for model checking.

AVANTSSAR [AVA] project is an ongoing project which is follow-on project of AVISPA project discussed earlier. AVANTSSAR investigates extensively the different adversary models for compromising data. It highlights some of the possible extensions to Dolev-Yao model. The XML injection attack discussed in the project focuses on the attack that a new node is inserted to the XML envelope by the intruder and does not include the attacks where additional information is added to the content of an XML message. When a node is inserted into an XML envelope it leads to some information being added to the message. Inserting a new node to the XML structure may go undetected by the destination service during parsing. Similarly, some content can be added to the SOAP message to change its meaning. Inserting some content in the message digest for signature or encryption will cause the message to fail authentication and will be rejected by the destination service during parsing. The models used in TSSAR project are independent of the security properties of a protocol and the channels explored by the project support different security properties such as confidentiality, authentication or both. However, once compromised, these channels can lead to complete control of the channel by the intruder. The main focus of their work is on securing channels by assigning them specific goals. Work described in

[BCFG04] is based on only securing the protocol run itself.

A leading trend for modelling Web services security protocols is to use a form of pi calculus. Other possible technologies for modelling and analysis of Web services based security protocols need to be investigated which may yield better modelling and analysis results. Moreover, there is a need for a feasibility analysis on the application of traditional model checking technologies to Web services protocols. So far, the focus has been on designing protocols and finding attacks on them. Another possible approach is to show the lack of attacks against protocols based on Web services specifications.

Recent works in the application of formal methods have been towards the modelling and improving of workflows in web services [pol], [pot]. [NB11] studies scenarios where the composition process may lead to failure due to incomplete specification of goal or unawareness on user's side of the functionality provide by the service. Kucukoguz [KS11] studies the modelling of artifact-centric data-aware workflow model. Fu [Fu11] proposed a logic based framework for formally specifying and reasoning about the implementation of privacy protection by a web application.

In other works, Christiansen [CC11] provides a first direct formalisation of the semantics of inclusive gateways described in Business Process Modelling Notation. Hee [HM11] concentrates on refinements of service composition using Petri nets. [ACN11] presents a methodology for passive testing based on invariants of distributed systems with time information. Properties under test are represented by invariants. When the tested system performs a requested task, the behaviour is reflected in the invariant. The invariants work

by checking the correctness of the logs recorded in each isolated system. Weidlich [WEW11] studies the behavioural consistency of process models representing different process perspectives. Formal models are being widely adopted for verifying not only the security but also the different aspects of web services such as composition and choreography.

The main objective of the modelling process has been to standardise Web service based security protocols so that they can be adopted widely. We believe that Web services security based protocols can be modelled in a better way using automaton theory. Automaton theory will allow us to model in detail the XML based messages. In this thesis Web services based protocols are modelled in terms of Dolev Yao abstractions. The cryptographic operators are considered symbolically. Fully automated verification techniques are applied to the abstractions. We investigate whether the existing formal verification techniques provide adequate modelling and analysis features for security protocols based on Web services.

We propose the innovative approach of using *Multistack Pushdown Automata* [HMU06] for modelling Web services based security protocols, where each service has a dedicated 'function' stack to be applied to the messages. We believe that such a model has more promising features and could easily detect any variations from the true model. We perform model building using pushdown automaton theory, and build the Web services based security protocols based on these automaton models using traditional cryptographic notations. The main benefit of using this approach is that the discussion can be extended beyond the Dolev-Yao model. As pointed out in Meadows' and Backes' work [Mea03, BG05], recent trends in network security infrastructure

may require the attack model to be made more robust. However, we argue that a more flexible system is required to incorporate such an attack model. We start our analysis by assuming the attack model to be Dolev-Yao based. However, after completing our preliminary analysis we intend to add an XML Injection Attack to the Dolev-Yao framework. The work in this thesis tries to address vulnerabilities in the area of WS-* stack security.

The contributions of this thesis are in the application of automaton theory to Web services based security protocols, in particular the novel approach to modelling WS-Security and WS-Trust by using pushdown automata. This thesis presents a transition system for the environment of these protocol runs. We believe that modelling using automaton theory makes the detection of variations to the XML based message easier. Also, finite state modelling is more suitable for modelling Web services as the state explosion problem can be controlled. State explosion is an exponential growth of state space in real world problems. Model checking tools are faced with the state explosion problem. There are some approaches to resolve this problem, such as symbolic algorithms, bounded model checking algorithms, partial order reduction, abstraction and counter-example guided abstraction refinement. In this thesis partial order reduction is used to control the state explosion problem. Our approach is also more sensitive to content tampering. We show that our model is able to detect XML injection attacks. To the best of our knowledge this is the first attempt to model WS-Security and WS-Trust based cryptographic protocols in Promela and Spin. Our preliminary approach allows the properties of the environment and the model to be defined in a more flexible and precise way using Linear Temporal Logic. We define authenticity and secrecy properties and present them in Linear Temporal

Logic [KM08]. We also present an intruder model based on the Dolev-Yao abstraction for WS-Security and WS-Trust based protocols in Promela. The intruder model is extended to encompass XML injection attacks. Spin and Promela form an effective analysis tool for cryptographic protocols and finite state systems. We adopted Promela for our web services based protocol model and Spin for verification for this purpose.

CHAPTER 3

System Model

> "Model checking is an automated technique that, given a finite state model of a system and a formal property, systematically checks whether this property holds for (a given state in) that model" [BK08].

3.1 Chapter Objectives

The aim of this chapter is to define a system model of protocols using WS-Security and WS-Trust. The chapter deal with two main topics: *system properties* and *modelling protocols.* The requirements will be modelled as linear temporal logic formulas and the system/protocol environment will be formalised as a state transition system. The combination of the property specification and system model will be input to the Spin model checker. The

environment will be modelled for each of the protocols described. This chapter lays the foundation for the following work on analysis.

3.2 Model Checking

3.2.1 Model Checking Process

Model checking is based on a collection of techniques for the automatic analysis of a system. The model checker takes as input a description of the system and the properties of that system. The system in most cases is defined as a finite state system and its properties are expressed as temporal logic formulas. The model checker verifies whether the properties hold or not. In most cases if a property does not hold the model checker gives a counterexample. In practice, the model of the system being analysed is approximate, thus the results are limited as well. Errors in the model may still remain after the verification.

When applying model checking to a system design, three main phases may be identified, as described in [BK08]:

1. Modelling Phase. The modelling phase consists of *(i)* modelling the system in a language acceptable to the model checker being used, *(ii)* quick simulations on the model, and *(iii)* using the property specification language to formalise the property to be checked.
2. Running Phase. The system is checked to see if the properties defined using the model checker hold.

3. Analysis Phase. This checks whether the properties specified are satisfied or not. Depending on the result, the model is then refined, the properties re-designed, and the process repeated.

Figure 3.1 gives an overview of the model checking approach. The requirements of the system under consideration are first identified and these requirements are then formalised in some property specification language. The system is then modelled in a language acceptable to the model checker. A combination of the model and the properties of the model are then fed into the model checker. The model checker outputs the results as 'satisfied' if no property is 'violated', or 'violated' if a property fails for the model.

To build any model for verification purposes there are recommendations that should be followed. Following these guidelines helps to model the system under consideration correctly. The system reflects its desired properties.

3.3 Modelling Cryptographic Protocols

Before our protocol model is defined, there are certain guidelines for cryptographic protocols that should be followed. Since Web services based security protocols are, in essence, XML based cryptographic specifications, we have tried to follow these guidelines in designing the system under consideration.

Figure 3.1: Schematic view of Model Checking approach

3.3.1 Guidelines for Modelling Cryptographic Protocols

When writing a suitable protocol three main factors must be taken into account: *the principles involved in designing the protocol*, *the goals of the protocol*, and *common attacks on the protocol.* These three criteria will now be briefly reviewed.

Principles for Designing Cryptographic Protocols

To design a good cryptographic protocol there are some principles found in the literature that should be satisfied [Sch95, Sti02, Sta95, For94]. The protocol must be efficient with no unnecessary encryption and no inclusion of unnecessary messages, for example, double encryption. Every message should say what it means: the interpretation of the message should depend only on its contents. It should be possible to write down a straightforward sentence describing what the message means. The conditions for a message to be acted upon should be clearly set out. If the identity of a principal is essential to the meaning of a message, it is prudent to mention the principal's name in the message. It should be clear why encryption is being done. When a principal signs material that has already been encrypted, it should not be inferred that the principal knows the content of the message. It is necessary to be clear about what properties are assumed for nonces. The use of a predictable quantity can serve in guaranteeing newness, through a challenge-response exchange, but it should be protected so that an intruder cannot simulate a challenge and later replay the response. If timestamps are used as freshness guarantees then the difference between local clocks at

various machines must be less than the allowable age of a message deemed to be valid. A key may have been used recently yet be quite old, and possibly compromised. If an encoding is used to present the meaning of a message, then it should be possible to tell which encoding is being used. The protocol designer should know which trust relations his protocol depends on, and why the dependence is necessary.

Goals of Cryptographic Protocols

Each protocol is designed with a particular goal in mind. There are various goals for security protocols, such as *(1)* security, *(2)* key establishment, *(3)* authentication, *(4)* key freshness, *(5)* key exclusivity where a key is only known to the principals of the protocols, and *(6)* good key where a key is both fresh and exclusive.

The goals for the protocol model put forward in this thesis are based on a combination of a cryptographic protocol and Web services goals. For the protocol model based on WS-Security and the Simple Message Exchange Protocol, the goal is to maintain *secrecy* and *authentication.* For the second protocol based on WS-Trust and the Security Token Protocol, the goal is not only secrecy and authentication, but also establishment of a *security context.* The protocol goals are modelled in terms of Linear Temporal Logic. A complete description is given in Section 3.7.

Attacks on Protocols

When defining goals for security protocols the hostile environment in which they will run should be kept in mind. Each protocol has a set of specific goals depending on the type of attacks it is most likely to face. When designing protocols, the types of attacks possible on the protocol must be considered. Traditional cryptographic protocols are faced with attacks such as eavesdropping, modification of messages, replay attack, man-in-middle attack, reflection attack, and so on. The effects of these attacks can be lessened with good encryption. In the case of attacks such as denial-of-service (DoS) and typing attacks, there is no particular solution. In DoS attacks, the server farms can be made to distribute traffic load when a DoS attack is detected. With an evolving protocol environment and the increasing complexity of protocols, there are more and more types of attack emerging. Web services based security protocols are similar to traditional protocols, but have an added complexity. These protocols are not only susceptible to traditional cryptographic attacks, but also have their own targeted attacks, such as:

- Forceful browsing, where the attack is aimed at discovering unpublicised Web services.
- Dictionary attacks, where an attacker attempts to break passwords to gain access.
- Parameter tampering, where data passed as parameters to a Web service is tampered with.
- XML injection attacks, where some information is added to alter the meaning of a message

To be able to widely deploy Web services, it is necessary to be able to defend against malicious attacks. Therefore, there is a need to develop and formalise intruder models for Web services based security protocols.

This thesis defines two such security protocols based on Web services specifications. The first of these is the Simple Message Exchange Protocol (SMEP), based on WS-Security. The goal of SMEP is to authenticate securely between two services. The second protocol, Security Token Protocol (STP), is developed using the WS-Trust specification. STP aims at reaching a secure agreement on a security context between two services while correctly authenticating them to each other. These protocols are defined in the next section.

3.4 Security Protocol Models

WS-Security and WS-Trust provide the syntax for a broad range of protocols, but do not define the protocols themselves. This lack of protocol definition provides flexibility for the end user. As a result, the security protocols need to be defined along with their goals. These goals have to be carefully specified and then validated. This section defines a protocol for a system employing only WS-Security. Later another protocol model based on WS-Trust will be included. A difficulty arises from modelling the XML based protocols in Promela, a language acceptable to the Spin model checker. Promela and Spin will be discussed in the next chapter.

3.4.1 Simple Message Exchange Protocol

The Simple Message Exchange Protocol (SMEP), a message exchange protocol employing the WS-Security specification [NKHBM04], will now be described. The goals of the protocol are that both participants are able to authenticate each other and the message is kept secret in the presence of an attacker, that is, secrecy and authentication. These goals can be accomplished by using a combination of XML Signature and XML Encryption. The message is kept secret by encrypting the contents of the message using XML Encryption and authentication is achieved by signing the message using XML Signature.

The protocol has two participants, service A and service B. Each participant is associated with a RSA key pair (a public and a private key). A's key pair is represented by (pk_A, sk_A), and likewise for B we have (pk_B, sk_B), where pk_A represents the public key of A and sk_A represents the secret or private key of A, and similarly for B. It is assumed that the public key of B is known to A and the public key of A is known to B prior to the protocol run. It is also assumed that the private keys of both A and B are kept secret and are only known to A and B.

Service A sends a request to service B, and B sends a response to A. The request message contains the timestamp (TS), the nonce (N_A), the identity of A, and REQ, which represents a request. TS, N_A and REQ are signed by A for purposes of authentication using RSA-SHA1 with sk_A, and for the purpose of secrecy the message is encrypted using a symmetric key, which in turn is encrypted using the public key of B, pk_B. B sends a response to A. It signs the response (RES, original message from A) using sk_B and encrypts the response using a fresh symmetric key under pk_A.

A simple authentication protocol should have a client A sending a message to B with sufficient information for authentication. The timestamp (TS) and nonce are used to verify that the message exchange is for the current traffic, i.e the message is not replayed from an older session. The value of TS changes with each message exchange, i.e., for each protocol step TS is assigned a fresh value. XML Signature is used to authenticate the sender's identity, and encryption is used for the purpose of secrecy. In the following, sign_Y(X) denotes the digest of the data X encrypted with the private key of entity Y.

(1) A → B : sign_A(TS, REQ, N_A) | pk_B(symmkey_A) | symmkey_A(TS, REQ, N_A), M_1
(2) B → A : sign_B(TS, RES, N_B) | pk_*A*(symmkey_*B*) | symmkey_B(TS, RES, N_B) | M_1

Step 1: The message, M_1, sent by service A has three parts. In the above "|" represents concatenation. Service A sends the data REQ to B, it signs the timestamp TS, REQ, and nonce of A using its secret key. Service A generates a symmetric key (to be used in subsequent communication with B) and encrypts this using the public key of B. Service A then encrypts TS, REQ, and N_A using the symmetric key.

Step 2: On receiving the message from A, service B uses its private key to extract the symmetric key. Service B then uses this symmetric key to decrypt the third part of the message, thereby extracting TS, REQ, and N_A. Service B then authenticates the identity of A by checking the signature, sign_A(TS,REQ,N_A), and checks the timestamp. After this service B sends

a response (RES) to A as a four-part message. It signs TS, RES, and N_B, where TS is a fresh timestamp, and then encrypts TS, RES, and N_B using the symmetric key (note that symmkey_A = symmkey_B). Service B also encrypts the symmetric key using the public key of A, which allows A to check that this is the same symmetric key that it originally sent to B. It also sends the original message M_1 that it received from Service A. Service A can compare this with the message it originally sent to allow the detection of message tampering.

Next a variation to the above protocol is presented with the aim of showing a simple message exchange between two services, which do not share symmetric keys. This initial protocol run will now be modelled.

(1) A → B : A, B, N_A, TS | pk_B(REQ, sign_A(A, N_A, REQ, TS))
(2) B → A : B, A, N_B, N_A, TS | pk_A(RES, sign_B(B, N_B, A, N_A, RES, TS))
(3) A → B : A, B, N_A, N_B, TS | pk_B(ACCEPT, sign_A(B, N_A, N_B, A, ACCEPT, TS))

In the first message exchange between service A and service B, the first part of the message, (A, B, N_A, TS), contains the identity of the sending service A, the identity of the intended receiving service B, a freshly generated random value (the nonce N_A), and a timestamp for the message, TS. The second part of the message, pk_B(REQ, sign_A(A, N_A, REQ, TS)), encrypts the data REQ, and a signature for A, N_A, REQ, TS, with the public key of B, pk_B. This signature is the digest of the data A, N_A, REQ, TS encrypted with the private key of A.

On receiving the message service B uses its private key to decrypt and

extract REQ and sign_A(A, N_A, REQ, TS), and checks the signature of A and the freshness of the timestamp. In the second step of the message exchange, service B then sends a message back to service A to indicate that it has authenticated service A. It returns (B, A, N_B, N_A, TS) which includes its identity B, the identity of the sending party A, its nonce N_B, the nonce it received from service A, N_A, and a newly generated timestamp, TS. The second part of the message, pk_A(RES, sign_B(B, N_B, A, N_A, RES, TS)), encrypts with the public key of A the data RES, and the signature of the identities, their nonces, the RES data, and the timestamp.

On receiving the message from service B, service A uses its private key to decrypt and extract RES and sign_B(B, N_B, A, N_A, RES, TS), and checks the signature of B and the freshness of the timestamp. If these are acceptable then in the last part of the message exchange, service A sends the following response to service B to indicate that it has authenticated service B. The first part of the message (A, B, N_A, N_B, TS) contains the identity of both the services, and the timestamp TS and nonce N_B it received from service B. In the second part of the message, pk_B(ACCEPT, sign_A(B, N_A, A, N_B, ACCEPT, TS)), service B encrypts the data ACCEPT and sign_A(B, N_A, N_B, A, ACCEPT, TS) with the public key of B. It should be noted that in this message the nonce of A, N_A, and the timestamp, TS, are freshly generated. After service B has received and accepted this message then services A and B are mutually authenticated.

The SMEP protocol is applied when service A sends a message to service B. The message contains the identity of A and its nonce, in the form of a UsernameToken, as used in WS-Security. The message also contains a times-

tamp token which provides a mechanism to determine the freshness of the message. The UsernameToken and REQ, representing some data, is signed by A for the purposes of authentication. The REQ and the signed content is then encrypted by the recipient's public key.

Service B processes the message sent by A as follows. It decrypts the message using its private key, and validates the message by confirming that the contents are signed by the sender A. This can be done by decrypting the message with the key information provided in the message and comparing the result with the original message. If the signature values match, the message is authentic. Once the message has been authenticated, it sends a response to A. The response contains its UsernameToken, consisting of the identity of B and its nonce, N_b. It also returns the sender information, i.e., the sender's name and nonce. It signs its UsernameToken, A and RES. It encrypts REQ and the signed information with the public key of sender A. A process the message as before. If all checks pass, A sends an ACCEPT response to B.

3.4.2 Security Token Protocol

WS-Trust provides a framework for issuing security tokens, renewing security tokens and brokering trust relationships [NGG+07]. WS-Trust depends on a Security Token Service, STS, which is a dedicated service for evaluating requests for tokens and issuing tokens. A *security token* (ST) is a collection of *claims*, where a claim is a statement made about a client, for example, name, identity, key, etc. A *Security Context* (SC) is a concept referring to an established authentication state and negotiated keys, and a Security Context

Token (SCT) is a tangible representation of the SC concept [Aea05].

A simple three-message exchange protocol for issuing a security token will now be presented. The messages are exchanged between sender A and a Security Token Service, STS. The goal of the protocol is to establish a security context between the two processes while maintaining secrecy and authenticity. A Security Context Token points to a shared context between a client and a Web service. Keys can be derived using the contents of the Security Context Token, which in turn are used to protect communication between participants. Readers are referred to WS-SecureConversation for a detailed description of the usage of security context tokens [Aea05].

Service A sends a Request Security Token (RST) to the Security Token Service (STS). Service A requests the issuance of a security token for communication between itself and the Security Token Service. The Security Token Service processes the request and, upon accepting it, agrees on a partial security context (partial SC). The Security Token Service responds with a Request Security Token Response (RSTR) token. After the acceptance of the RSTR, both parties should agree on a full security context (SC). WS-Security is used for protection of envelopes carrying requests for security token elements and Request Security Token response elements.

(1) A → STS : A, N_A, STS, RST, TS | sign_A(A, N_A, RST, tokenType, requestType, appliesTo, clientEntropy, entropicMode) |pk_STS(N_A, RST, tokenType, requestType, appliesTo, clientEntropy, entropicMode)

(2) STS → A : A, N_A, STS, N_STS, RSTR, TS | sign_STS(RSTR, SC, SCT_ID, tokenType, requestType, appliesTo, serverEntropy, clientEntropy,

entropicMode,created, expires, ComputedKey) | pk_A(N_A, N_STS, SC, SCT_ID, tokenType, requestType, appliesTo, serverEntropy, clientEntropy , entropicMode, created, expires, ComputedKey)

(3) A → STS : SCT_ID | sign_A(SCT_ID) | pk_STS(SCT_ID)

The first message is sent from service A to the Security Token Service. The request security token consists of three parts. The first part of the message, A, N_A, STS, RST, TS, contains the identity of A, nonce N_A , the identity of the service it wants to talk to, the type of the message (RST) and the timestamp (TS). The second part of the message contains signed information sign_A(A, N_A, RST, tokenType, requestType, appliesTo, clientEntropy, entropicMode). Service A signs its identity, nonce and request (RST). It also signs all the elements of the Request Security Token including, TokenType (the type of token being requested), RequestType (request for issuance of security token), AppliesTo (the service where it will be used for communication), clientEntropy (its base64 encoded value) and entropicMode (the mode of calculating the entropy). In the last part of the message, pk_STS(N_A, RST, tokenType, requestType, appliesTo, clientEntropy, entropicMode), it encrypts the information with the Security Token Service's public key.

The second message is the request security token response (RSTR), which is the response from the Security Token Service after it has validated the request. The first part of the message, A, N_A, STS, N_STS, RSTR, TS, contains the identity and nonce information of A. It also contains the Security Token Service provider identity and nonce, STS and N_STS. The second part of the message, sign_STS(RSTR, SC, SCT_ID, tokenType, requestType, appliesTo, serverEntropy, clientEntropy, entropicMode, created, expires, Com-

putedKey), contains the signed information. It signs the information that was present in the original request, TokenType, AppliesTo, RequestType, clientEntropy and entropicMode. It now also returns additional information about calculating the keys. It sends its entropy, serverEntropy, and how the key is to be computed in the ComputedKey element. It also returns the unique security context id, SCT_ID. In the last part of the message, pk_A(N_A, N_sts, SC, SCT_ID, tokenType, requestType, appliesTo, server-Entropy, clientEntropy, entropicMode, created, expires, ComputedKey), the Security Token Service encrypts all the information signed by it before, and sends the message to the requestor.

The last message is sent by the service A to STS. It contains the unique identity of the security token. It signs the SCT_ID, sign_A {SCT_ID}, and encrypts it, pk_STS(SCT_ID) with the public key of the Security Token Service.

Both the client and the Security Token Service include a fresh random value called *entropy*, which is used to calculate the key. This key is used to establish a context key between services and is used as a session key. A session key is used when two services are involved in a session with each other. The session key is shared between these two services for encrypting the messages exchanged between them.

Service A sends a message to the Security Token Service. The message is authenticated using the mechanism established in SMEP and the request for a security token will be accepted only if STS knows it is talking to A and has a fresh N_A. After the STS validates the request for a security token, there should be a partial agreement represented by *partialSC*. *partialSC* consists

of *appliesTo*, *entropicMode*, *tokenType* and *requestType*. Here, the partial entropy mode is being used, where the requester and the Security Token Service both provide entropy to calculate the context key. The Security Token Service sends a response to service A, and A validates the message. When the message Request Security Token Response (RSTR) is accepted both processes should have agreed on a full security context. This Security Context contains an identity for the security context token, *SCT_ID*, which is unique to the Security Context and is known to parties using the Security Context, *clientEntropy* and *serverEntropy*, and *computerKey*, *entropicMode*, *appliesTo*, *tokenType*, *requestType* and *expires* are the other elements contained in the Request Security Token Response element.

Figure 3.2 gives a graphical representation of the Security Token Protocol. A sender service sends a request for a security token (RST) envelope. The Security Token Service processes the request and, if successful, returns the requested security token (RSTR).

3.5 Environment Model for Protocols

When defining a protocol, the environment in which the protocol will run must be considered. This environment needs to be represented formally so it can not only be easily mapped into a model checking framework, but also reflects accurately the environmental context. In this thesis the environment is modelled as a *transition system*, defining the changes involved in each part of the protocol run.

Transition systems are often used to describe the behaviour of a system,

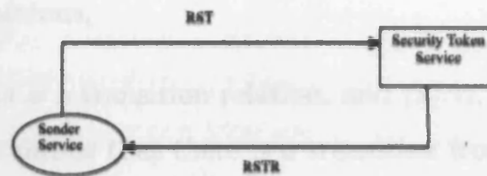


Figure 3.2: Security Token Protocol

and are defined with *action names* for the transitions and *atomic propositions* for the states. An atomic proposition (AP) is one whose truth or falsity does not depend on the truth or falsity of any other proposition. Action names are used for describing communication mechanisms between processes. Atomic propositions express simple known facts about the state of the system and are used to formalise temporal characteristics. A simple choice is to let the state names act as AP, i.e., $L(s) = s$ for any state s. The equation 3.1 describes a transition system.

$$TS = (s, Act, \rightarrow, I, AP, L) \tag{3.1}$$

where,

s is a set of states,

Act is a set of actions,

$\rightarrow \subseteqq s \times Act \times s$ is a transition relation, and $(s_1, \alpha, s_2) \in \rightarrow$ is denoted by $s_1 \overrightarrow{\alpha} s_2$. This means that there is a transition from state s_1 to state s_2 associated with action α.

$I \subseteqq s$ is a set of initial states,

AP is a set of atomic propositions,

$L: s \rightarrow 2_{AP}$ is a labelling function that relates a set $L(s) \in 2_{AP}$ of atomic propositions to any state s.

$L(s)$ stands for exactly those atomic propositions $a \in AP$ which are satisfied by state s. Given Φ is a propositional logic formula, then s satisfies the formula Φ if the evaluation induced by $L(s)$ makes the formula Φ true, that is

$$s \models \Phi \iff L(s) \models \Phi$$

where $s \models \Phi$ should be read as "the state s models the propositional logic formula Φ". This means that when all the atomic propositions for a state s can be obtained from the propositional logic formula Φ then we say that s satisfies/ formula Φ, and *vice versa.*

3.5.1 Environment Model for SMEP

The model for the environment of the Simple Message Exchange Protocol will now be given. The principal services involved in the environment, the

sender, receiver, and intruder services, and all possible transitions between the services are presented. The environment is described as a transition system.

A finite state transition model of the Simple Message Authentication Protocol, or SMEP consists of six states representing the protocol run at various stages. The sender sends a message to the receiver over an insecure channel. The message is encrypted at the sender's side before being sent on the channel, and is decrypted at the receiver's end. The intruder can listen to these messages and can replay them or start their own conversation. The sender and receiver roles can be adopted by service A or B, depending on which service is initiating a protocol run.

Below a transition system for the Simple Message Exchange Protocol is described. Equation 3.3 for SMEP represents the complete transition of the system as described in Eq. 3.1. S denotes the Sender Service and R the Receiver Service. IC is the insecure channel on which the message is passed.

$$\begin{aligned}SMEP \;=\; & ((S, Sign/Encrypt, IC, Intruder, Decrypt/VerifySign, R),\\ & (encrypt_msg, decrypt_msg, send_msg, recv_msg, listen_msg),\\ & (S \xrightarrow{encrypt_msg} Sign/Encrypt), (Sign/Encrypt \xrightarrow{send_msg} IC),\\ & (IC \xrightarrow{decrypt_msg} Decrypt/VerifySign), (Decrypt/VerifySign \xrightarrow{recv_msg} R),\\ & (I \xrightarrow{send_msg} IC), (IC \xrightarrow{send_msg} I), S,\\ & (secrecy, authentication),\\ & (L(S) = \{\phi\}, L(IC) = \{\phi\}, L(Intruder) = \{\phi\}, L(R) = \{\phi\},\\ & L(Sign/Encrypt) = \{\phi\}, L(Decrypt/VerifySign) = \{secrecy, authentication\}) \end{aligned} \tag{3.3}$$

In the above the set of states is {S, Sign/Encrypt, IC, I, Decrypt/VerifySign, R}, where:

1. S: waiting for the sender to take some action.
2. R: waiting for the receiver to take some action.
3. I: waiting for the intruder to take some action.
4. Encrypt/Sign: represents the state when the message is encrypted and signed.
5. Decrypt/VerifySign: represents the state when the message has been received, but has not yet had its signature verified or been decrypted.
6. IC: represents the state when the message is travelling on the insecure channel.

In a Simple Message Exchange Protocol run the set of possible actions is {encrypt_msg, decrypt_msg, send_msg, recv_msg, listen_msg}. ‘encrypt_msg’

represents the encryption of the message, 'decrypt_msg' represents the decryption of the message, 'send_msg' and 'recv_msg' symbolize sending and receiving of messages between the sender service and the receiver service, and 'listen_msg' represents the Intruder action of listening on the message channel and intercepting the messages.

The set of atomic propositions of a system represents simple facts about the system, and are used for formalizing the system properties. In their simplest form, they can be represented by the name of the states in the protocol environment model. The atomic propositions for the environment of the Simple Message Exchange Protocol are 'Secrecy' and 'Authentication'. 'Secrecy' and 'Authentication' illustrate that the message has been kept secret during the run over the channel by means of encryption and 'Authentication' refers to verification of the sender's identity by confirming that the message was correctly signed by the sender service. Thus, the set of atomic propositions is {Secrecy, Authentication}.

The set of transitions is:

- S $\overrightarrow{\textit{encrypt_msg}}$ Sign/Encrypt,
- Sign/Encrypt $\overrightarrow{\textit{send_msg}}$ IC,
- IC $\overrightarrow{\textit{decrypt_msg}}$ Decrypt/VerifySign,
- Decrypt/VerifySign $\overrightarrow{\textit{recv_msg}}$ R,
- I $\overrightarrow{\textit{send_msg}}$ IC),
- IC $\overrightarrow{\textit{send_msg}}$ I,

For example, the first of these represents the transition of the system from state S to state Sign/Encrypt caused by the action encrypt_msg.

$L: S \rightarrow 2_{AP}$ is a labelling function, which defines the properties that a state is supposed to satisfy. L(S)={ϕ}, L(IC)={ϕ}, L(I)={ϕ}, L(Sign/Encrypt) = {ϕ} and L(R)={ϕ} give the states which do not need to satisfy any atomic propositions. L(Decrypt/VerifySign) = {secrecy, authentication} represents the state 'Decrypt/VerifySign' which must satisfy the properties of 'Secrecy' and 'Authentication'.

3.5.2 Environment Model for STP

This section models the environment for the Security Token Protocol (STP). The environment model is built on top of the one for SMEP described in the previous section, but the complexity of the environment for issuing security tokens to Web services is added. A sender service, a Security Token Service and an intruder communicate over an insecure channel. The sender service requests the Security Token Service to issue a security token. The request for the security token is created, the resulting message is signed and encrypted, and sent over the insecure channel IC. On the receiver end the request is processed and the Security Token Service agrees on a partial security context. It then sends its complete information for calculating the security context back to the sender over the same insecure channel. S denotes the sender service, I the intruder, and STS the Security Token Service service. IC is the insecure channel on which the message is passed. Equation 3.4 symbolizes the transition system of the Security Token Protocol environment.

$$
\begin{aligned}
STP = \; & ((S, RST, RST_Sign/Encrypt, IC, RST_Decrypt/VerifySign, Partial_SC, STS, \\
& RSTR, RSTR_Sign/Encrypt, RSTR_Decrypt/VerifySign, Full_SC, I), \\
& (create_RST, Enc_RST, Send_RST, Recv_RST, Decrypt_RST, Partial_SC, \\
& Create_RSTR, Enc_RSTR, Send_RSTR, Recv_RSTR, Decrypt_RSTR, SC_RST) \\
& ((S \; \overrightarrow{Create_RST} \; RST), (RST \; \overrightarrow{Enc_RST} \; RST_Sign/Encrypt), \\
& (RST_Sign/Encrypt \; \overrightarrow{Send_RST} \; IC), (IC \; \overrightarrow{Recv_RST} \; RST_Decrypt/VerifySign), \\
& (RST_Decrypt/VerifySign \; \overrightarrow{Decrypt/VerifySign_RST} \; Partial_SC), \\
& (Partial_SC \; \overrightarrow{PartialSC_RST} \; STS), (STS \; \overrightarrow{Create_RSTR} \; RSTR), \\
& (RSTR \; \overrightarrow{Enc_RSTR} \; RSTR_Sign/Encrypt), (RSTR_Sign/Encrypt \; \overrightarrow{Send_RSTR} \; IC), \\
& (IC \; \overrightarrow{Recv_RSTR} \; RSTR_Decrypt/VerifySign), \\
& (RSTR_Decrypt/VerifySign \; \overrightarrow{Decrypt/VerifySign_RSTR} \; Full_SC), \\
& (Full_SC \; \overrightarrow{SC_RSTR} \; S)), \{S\}, \\
& (Secrecy, Authentication, partialSC, SC), \\
& (L(S) = \{\phi\}, \; L(RST) = \{\phi\}, \; L(RST_Sign/Encrypt) = \{\phi\}, \\
& L(IC) = \{\phi\}, \; L(I) = \{\phi\}, \; L(RST_Decrypt/VerifySign) = \{Secrecy, Authentication\}, \\
& L(Partial_SC) = \{partialSC\}, \; L(STS) = \{\phi\}, \; L(RSTR) = \{\phi\}, \\
& L(RSTR_Sign/Encrypt) = \{\phi\}, \; L(RSTR_Decrypt/VerifySign) = \{Secrecy, Authentication\} \\
& L(Full_SC) = \{SC\})) \qquad (3.4)
\end{aligned}
$$

In the above specification of the Security Token Protocol the set of states is as follows:

1. S: waiting for the sender to take some action.
2. RST: represents the state when the request for a security token has been generated.
3. RST_Sign/Encrypt: represents the state when the request for a security token has been encrypted and signed.

4. IC: represents the state when the message is travelling on the insecure channel.

5. RST_Decrypt/VerifySign: represents the state when the request for a security token has been received, but has not yet had its signature verified or been decrypted

6. Partial_SC: represents the state when a partial security context has been established between the sender and the Security Token Service. This means the Security Token Service has authenticated the sender and accepted their request for a security token, but has not yet responded to establish a full security context.

7. STS: waiting for the Security Token Service to take some action.

8. RSTR: represents the state when the response to the security token request has been generated.

9. RSTR_Sign/Encrypt: represents the state when the response to the security token request has been encrypted and signed.

10. RSTR_Decrypt/VerifySign: represents the state when the response to the security token request has been received by the sender, but has not yet had its signature verified or been decrypted.

11. Full_SC: represents the state when a full security context has been established between the sender and the Security Token Service.

12. I: waiting for the intruder to take some action.

In a Security Token Protocol run the set of possible actions is {Create_RST , Enc_RST, Send_RST, Recv_RST, Decrypt_RST, Partial_SC, Create_RSTR,

Enc_RSTR, Send_RSTR, Recv_RSTR, Decrypt_RSTR, SC_RST)}. 'Create_RST', 'Enc_RST' and 'Send_RST' represent actions which are performed on the sender side when creating, signing, encrypting, and sending a request for a security token. 'Recv_RST', 'Decrypt_RST' and 'Partial_SC' denotes actions on the STS end of the request for a security token message. The request is received, decrypted, the signature verified and an agreement is made on a partial security context. The Security Token Service then performs the following actions to create, sign, encrypt and send a request security token response message: 'Create_RSTR', 'Enc_RSTR' and 'Send_RSTR'. The request security response is received by the sender: 'Recv_RSTR'. It is then decrypted and its signature is verified by the action 'Decrypt_RSTR'. A security context is then established by the sender represented by the following action: 'SC_RST'.

The set of atomic propositions for the Security Token Protocol environment is {Secrecy, Authentication, partialSC, SC}. 'Secrecy' and 'Authentication' represent that the request security token message and the request security token response message are kept encrypted and are digitally signed. 'partialSC' denotes the property of agreement on a partial security context by the Security Token Service, i.e., when the request for a security token is acceptable to the Security Token Service. 'SC' shows the property that the security context has been agreed on by the sender. This property is fulfilled when the sender has agreed on the security token provided in the request security token response message.

The set of transitions is:

- S $\xrightarrow{Create_RST}$ RST,

- RST $\overrightarrow{Enc_RST}$ RST_Sign/Encrypt,
- RST_Sign/Encrypt $\overrightarrow{Send_RST}$ IC,
- IC $\overrightarrow{Recv_RST}$ RST_Decrypt/VerifySign,
- RST_Decrypt/VerifySign $\overrightarrow{Decrypt/VerifySign_RST}$ Partial_SC),
- Partial_SC $\overrightarrow{PartialSC_RST}$ STS,
- STS $\overrightarrow{Create_RSTR}$ RSTR,
- RSTR $\overrightarrow{Enc_RSTR}$ RSTR_Sign/Encrypt,
- RSTR_Sign/Encrypt $\overrightarrow{Send_RSTR}$ IC,
- IC $\overrightarrow{Recv_RSTR}$ RSTR_Decrypt/VerifySign,
- RSTR_Decrypt/VerifySign $\overrightarrow{Decrypt/VerifySign_RSTR}$ Full_SC),
- Full_SC $\overrightarrow{SC_RSTR}$ S

For example, the transition 'S $\overrightarrow{Create_RST}$ RST' represents a transition from state S to state RST when action Create_RST is performed.

L is a labelling function, which defines the AP properties that a state has to satisfy. L(S) = $\{\phi\}$, L(RST) = $\{\phi\}$, L(RST_Sign/Encrypt) = $\{\phi\}$, L(IC) = $\{\phi\}$, L(I) = $\{\phi\}$, L(STS) = $\{\phi\}$, L(RSTR) = $\{\phi\}$, L(RSTR_Sign/Encrypt) = $\{\phi\}$ } represents that the states do not need to satisfy the atomic propositions described above. L(RST_Decrypt/VerifySign) = {Secrecy, Authentication}, L(Partial_SC) = {partialSC}, L(RSTR_Decrypt/VerifySign) = {Secrecy, Authentication}, L(Full_SC) = {SC} } denotes the states which satisfy atomic propositions. L(RST_Decrypt/VerifySign) = {Secrecy, Authentication} means

that the state 'RST_Decrypt/VerifySign' should satisfy the property of 'Secrecy' and 'Authentication'. Similarly, L(Full_SC) = {SC} means that the state 'Full_SC' satisfies the property of 'SC', meaning an agreement has been reached on a security context.

We also assume that during a protocol run there is no message loss, the message is delivered to the end point correctly.

The preceding discussion has established environment models for the SMEP and STP protocols, and also has given a formal description of the possible states and transitions involved in each environment. The properties of the intruder, and the possible ways an intruder can interact with the environment, will now be discussed in detail.

3.6 Intruder Model for Protocols

This section defines the properties of the intruder and the attack model. It is assumed that the attacker in the model has abilities as specified by the Dolev-Yao model. The ability to:

- Overhear and intercept all the messages on the network.
- Modify the messages.
- Generate new messages using the information from overheard messages and some beforehand information.
- Send a new or captured message to another entity in the system.

In addition, it is assumed that the underlying cryptography cannot be broken.

The behaviour of the intruder can be modelled in two ways: *(1)* the intruder intercepts messages, and *(ii)* the intruder sends messages. In the second case the intruder can send two types of messages on the network. He or she can replay an old message, or create a new message from information learned so far.

3.6.1 Manipulated Protocol Run for SMEP

There are four possible ways the intruder can take part in the protocol session and interact with the participants. Here, $\rightarrow$ represents initiation of a protocol run from the service at right to left, i.e, A$\rightarrow$B is read as A initiates a conversation with service B.

I $\rightarrow$ B : The intruder I behaves like a legitimate user of the system. S/he initiates a session with B and sends a message to B. The aim of the intruder is to learn as much information as possible from service B. In this run, the intruder is able to learn the nonce, a randomly generated value from B. S/he can use this nonce to initiate message exchange with service A, acting as B.

A $\rightarrow$ I : A talks to I. Service A assumes I is a legitimate service. Intruder service I completes a successful run with the A. I learns as much information as possible. Service I learns the nonce of A and can use it to pretend to be A.

A $\rightarrow$ I(B) : A starts a message exchange with service B. However, the intruder service intercepts the message and pretends to be service B itself. I

can only do this once it knows the nonce of B, which has not been used in a message exchange before between service A and service B.

I(A) $\rightarrow$ B : Service I initiates a run with service B, pretending to be service A. It can only start a message exchange with service B if it has access to the nonce of A, which has not been used before in any exchange between service A and service B.

The intruder can initiate a protocol run with service A and service B. During these runs the intruder learns the nonces for A and B. It can use these to initiate further runs as an impostor. Protocol runs being executed after it learns the nonces are $A \rightarrow I(B)$ and $I(A) \rightarrow B$.

3.6.2 Manipulated Protocol Run for STP

This section defines the possible interactions of the intruder service with the Security Token Service. There are two possible scenarios where an intruder can interact with the Security Token Service.

$I \rightarrow STS$: The intruder acts as a legitimate user of the environment. The intruder initiates a message exchange with the Security Token Service and requests a security token used for establishing a session with another service. The Security Token Service believes service I to be a legitimate user and issues a security token to it.

$I(A) \rightarrow STS$: Service I can initiate a message exchange with the Security Token Service pretending to be service A. The intruder may have gained

information from a previous interaction with service A. It can use the information, such as the nonce of A, to act as A and gain a security token. Service I behaves as an impostor.

3.7 System Properties

This last two sub-sections have defined the capabilities of the intruder for both the Simple Message Exchange Protocol and Security Token Protocol. The possible ways the protocols can interact with the environment have been described. This section describes the properties of the protocols in the form of Linear Temporal Logic.

The behaviour of a system may be modelled as formulas in Linear Temporal Logic (LTL). Temporal Logic is the branch of logic which allows one to reason about the causal and temporal relations of properties [Hol03]. The properties of a protocol run can be formalized unambiguously and concisely with the help of temporal operators. Linear Temporal Logic is a dominant formalism in verification, and can be applied to finite and infinite runs of a system.

The next two subsections define the LTL operators used in defining system properties. The square operator, $\Box p$, defines that a property p will remain true throughout a run. The operator $\Box$ is read as *always.* The diamond operator, $\Diamond p$, defines that the property p is guaranteed to become true at least once in a run. The operator $\Diamond$ is read as *eventually.* **X** is the next operator, and **X**(p) is read as "Next p". $\rightarrow$ represents the boolean operator

for logical implication. ! represents a logical operator for negation.

3.7.1 Property Specification for SMEP

The goal of the Simple Message Exchange Protocol, as discussed earlier, is authentication and secrecy. If A talks to B and is satisfied that it is communicating with B, and B is satisfied that it is communicating with A, and, if intruder I has not learned the nonce of A or B, then it is said that A and B have successfully completed a run of the protocol. This property of the system can be modelled in temporal logic. A well-formed temporal formula is a combination of state formulae and temporal operators. This formula is input to the Spin model checker, along with the system model. The authentication between A and B can be modelled as shown below.

statusA → !nonceB
statusB → !nonceA
X(◇ SenderBindAB → □ (SenderBindAB ∧ RecvrChallengeAB))
X(◇ RecvrBindAB → □ (RecvrBindAB ∧ SenderChallengeAB))

The global LTL variables represents,

- statusA : is an LTL variable that is updated when service A does not know nonce of B. Its value is changed at the start of the protocol run.

- nonceB : represents the knowledge of nonce of B by service A.

- statusB : Is updated when service B initially does not know nonce of A. It represents the knowledge of nonce of A by service B

- nonceA: It represents the knowledge if a nonce by Service B

- SenderBindAB : A global LTL variable updated by a function executed by service A when it is ready to commit to service B

- RecvrChallengeAB : A global LTL variable updated at the receiver end(service B) when it knows it is talking to service A. This is achieved by verifying the identity of service A.

- SenderChallengeAB: Global variable which is updated by a function at the sender end when it starts a protocol run with service A

- RecvrBindAB : is updated when service B is ready to commit to service A. The variable is updated when the protocol satisfies the goals, i.e authentication and confidentiality

Variables nonceB, nonceA, SenderBindAB, RecvrBindAB, SenderChallengeAB and RecvrChallengeAB are global variables used for modelling the properties in temporal logic, and are input to the model checker during verification of the protocols. Initially, A and B do not know the nonces of each other. The *SenderBindAB* variable eventually becomes true when *SenderBindAB* and *RecvrChallengeAB* are always true. *RecvrChallengeAB*

becomes true only when B is talking to A and *SenderBindAB* becomes true when A commits to a session with B. Next *RecvrBindAB* eventually becomes true when *RecvrBindAB* and *SenderChallengeAB* are always true. *SenderChallengeAB* becomes true when A knows it is talking to B and *RecvrBindAB* becomes true when B commits to a session with A. At the end of the run, A and B will know the nonces of each other.

3.7.2 Property Specification for STP

The goal of the protocol is to exchange a Security Context between a sender service *A* and a Security Token Service *STS*. A security context establishes an authenticated state between the two services and negotiated keys which have additional security properties. SenderBindAS, SenderChallengeAS, RecvrChallengeAS, SenderChallengeAS are global variables used in Promela, a modelling language for Spin. The specification properties for the Security Token Protocol are modelled as follows

($\Diamond$ SenderBindAS $\rightarrow \Box$ (SenderBindAS $\wedge$ RecvrChallengeAS)
X($\Diamond$ RecvrBindAS $\rightarrow \Box$ (RecvrBindAS $\wedge$ SenderChallengeAS)
X($\Diamond$ PartialSC $\rightarrow \Box$ (AppliesTo $\wedge$ TokenType $\wedge$ RequestType $\wedge$
EntropicMode $\wedge$ clientEntropy)
X($\Diamond$ SC $\rightarrow \Box$ (partialSC $\wedge$ serverEntropy $\wedge$ ComputedKey $\wedge$ expires $\wedge$
SCID)

The global LTL variables represents,

- PartialSC : is an LTL variable updated when security token service(sts) agrees with the information sent by the requestor for requesting for a security token.
- AppliesTo: is updates if the sts agrees service the token the applicable for.
- TokenType : is updated when the type of token being requested is valid, i.e, security context token.
- RequestType : is updated when it is a request for issuing a security token.
- EntropicMode: is updated when both services agree to use partial entropy. This is used for calculating keys.
- clientEntropy : represents the entropic value provided by the client for calculating the keys.
- SC: represents the security context which is used by the requestor to establish a session with the desired service.
- serverEntropy : represents the value provided by the server to calculate the keys.
- ComputedKey : represents how the keys will be computed and is updated accordingly.
- expires: represents how long the security context token is valid for.
- SCID: is the unique ID for the security context token.

The authentication properties must be satisfied first, as explained in Section 3.7.1. Next 'PartialSC' eventually will be true when the Security Token

Service STS agrees on 'AppliesTo' (which service the token is valid for), and 'TokenType' (the type of token being requested). 'RequestType' defines what is requested by the initiator, e.g., a request for issuing a security context token. 'EntropicMode' defines whether both the initiator and the service provider will be providing entropies to compute a key used by the sender, and 'clientEntropy' is the entropic value provided by the client.

Next 'SC' will eventually be true when there is an agreement on 'partialSC', 'serverEntropy' (provided by the server), 'ComputedKey' (tells how the key is to be computed), 'expires' (when the token is valid till), and the 'SCID' (the unique identifier for the security context token).

3.8 Concluding Remarks

This chapter has presented the building blocks for the model to be used subsequently. Two protocols have been defined: the Simple Message Exchange Protocol (SMEP) and the Security Token Protocol (STP). The goals for each protocol have been stated: SMEP aims to achieve authentication and secrecy, whereas STP aims to establish a security context. The hostile environment for both the protocols are modelled as transition systems, in which all possible principals and transitions involved in the protocol run are defined. The correctness properties for each protocol have been defined in terms of Linear Temporal Logic. These LTL properties can be used to verify the correctness requirements for the protocols. The next chapter defines the XML envelopes for protocols SMEP and STP as multi-stack pushdown automata. Each automaton captures the detailed work of the protocol run and the functions

applicable to the XML elements of the messages. We believe such a model can be more beneficial in detecting XML based attacks. An XML injection attack will also be modelled, and simulations run against our model. The automata are modelled with the modelling language Promela and analyzed using the Spin model checker.

These protocols can be extended over multiple services, and each service will be allocated its own stack. The protocol are blocking and synchronous. The services will wait for the response after a request before they can move on to the next run. The stack provides unlimited memory, this will allow us to model complex protocols with increased functionality. However, increasing the number of states in the model may lead to state explosion problem. A possible solution is to divide the protocols in subsets and then analyse their working separately. SMEP and STP are mainly designed for two services. In the situation where multiple services are involved, there will be no extra impact on the stacks as each service has its own stack. In the case of SMEP, multiple services involved in the protocol run can be classified as sending and receiving services. For example, suppose we have four services involved in a protocol run A,B, C and D. We will get a the following set of sending and receiving services {A,B}, {B,C},{C,D}. In the first set, A is the sending service and B is the receiving service, then in the second set, B becomes the sending scrvicc and C thc rccciving scrvicc and finally, in thc last sct, C the sending service and D is the receiving service. We describe two protocols based on WS-Security(Simple Message Exchange Protocol) and WS-Trust(Security Token Protocol). These protocols were selected as they represent the simplest functionality between two services, sending and receiving messages and requesting a security token from a security token service. They can be later

extended to encompass more complexity.

Works described in [Bla02] [TCCD07] [KR05][GP02][BF04] focusses on modelling the functionality/working of the WS-* based security protocols. Our work allows modelling the functionality and the goals of protocols in a single model.

In practice, our approach will allow the end users to analyse the correctness of a large security system during simulation runs instead of verifying the correctness in the later verification phase.

We use push-down automata to model WSBCPs as the combination of the input tape, automaton and the stack allows us to capture the behaviour and properties of the protocols. The PKI standards are employed by WS-* during protocol runs for the purpose of security, e.g. WS-Trust employs X.509 certificates for the purpose of authentication. It is used by the services to validate the identity of the service represented by the certificate.

CHAPTER 4

Modelling Protocols with Automata

Chapter 3 presented two protocols models: the Simple Message Exchange Protocol based on WS-Security, and the Security Token Protocol based on WS-Trust. The goals of the Simple Message Exchange Protocol were defined as authentication and secrecy, and the goal of the Security Token Protocol is the establishment of a security context. In Chapter 3 the hostile environment was modelled for both the protocols as a transition system. Linear Temporal Logic was used to define the goals of the protocols.

4.1 Chapter Objectives

This chapter presents pushdown automata for the Simple Message Exchange Protocol and Security Token Protocol defined in Section 3.4. These models

reflect the behaviour of the protocols and the properties they are supposed to satisfy. The Simple Message Exchange Protocol pushdown automaton satisfies the properties of authentication and secrecy. The Security Token Protocol automaton satisfies the property of establishing a security context.

4.2 Pushdown automaton

A formal notation of the two-stack pushdown automata (PDA) model will now be given. Note that the definition can be extended to include more stacks. A two-stack PDA is defined by the following notation:

$$M = \left(Q, \sum, \tau, \delta, q_0, y, z, F\right) \tag{4.1}$$

where

Q is a finite set of internal states of the control unit,

$\sum$ is the input alphabet,

τ is a finite set of symbols called the stack alphabet,

δ is a mapping of $Q \times (\sum \cup \{\lambda\}) \times \tau \times \tau$ to finite subsets of $(Q \times \tau^* \times \tau^*)$. δ is called the transition function, λ denotes teh empty string, and τ^* is a finite list of elements in τ.

$q_0 \in Q$ is the initial state of the control unit,

$y \in \tau$ is the first stack start symbol,

$z \in \tau$ is the second stack start symbol,

$F \subseteq Q$ is the set of final states.

Every transition from one state to another in the automaton is made by observing both stacks. Initially, the system is in state q_0 and both stacks have symbols y and z at the top. The stack language τ contains stack symbols (stack alphabets will be treated as strings rather than as alphabets). If the final state is reached with both the stacks empty, the input is accepted as valid. A transition function, δ, is represented as $\{\sum$, pop τ, pop τ, push τ, push $\tau\}$. The first pair of pop and push applies to the first stack, and the second pair of pop and push applies to the second stack in the PDA. In order to keep the discussion focused on the problem at hand, minor details of the workings of a PDA are omitted here. For more details readers are referred to [Lin06].

A two-stack pushdown automaton for modelling both the Simple Message Exchange Protocol and Security Token Protocol will now be presented. In the model, each participant service is assigned a stack, e.g., service A is assigned stack Service A. Each stack contains the functions necessary for completing a protocol run.

4.3 Simple Message Exchange Protocol

The Simple Message Exchange Protocol (SMEP), defined in Section 3.4, exchanges messages between two services, service A and service B. The goals of the protocol are secrecy and authentication, as explained in Chapter 3. SMEP is modelled as a pushdown automaton to reflect these goals. Figure 4.1 illustrates the pushdown automaton model for SMEP, and Table 4.1 shows the stacks used in the model.

A pushdown automaton reads information from a tape and executes a transition from one state to another based on the information read from the tape. The input alphabet of the tape represents actions which are to be conducted during message exchange between service A and service B in the Simple Message Exchange Protocol. The steps involved in a complete run of the Simple Message Exchange Protocol are encoded as input alphabet symbols of the tape. The input alphabet symbols for the complete consumption of a message between a sender and receiver service for the Simple Message Exchange Protocol is { sChallenge, TS, SIGN, ENCRYPT, Send/Recv_Msg, DECRYPT, TS, vSIGN, RecvChal, TS, SIGN, ENCRYPT, Send/Recv_Msg, DECRYPT, TS, SIGN, sBIND, TS, SIGN, ENCRYPT, Send/Recv_Msg, DECRYPT, TS, vSIGN, rBIND}. These input variables illustrate steps involved during a message exchange in a single protocol run between the sender and receiver processes. When each alphabet symbol is read some process is executed from the stacks of the sender and receiver processes. 'sChallenge' represents the sender service initiating a run with the receiver service and updating the sender challenge variable. 'TS' stands for the allocation of a timestamp value to a message to guarantee freshness. 'SIGN' represents initiating the signing of the message and embedding the signature information needed to validate

it. 'ENCRYPT' represents integrating the necessary encryption information in the message and encrypting the message with the receiver's public key. 'Send/Recv_Msg' represents the act of sending or receiving the message on the insecure channel. 'DECRYPT' represents the decryption process at the receiver's end using its private key. 'TS' represents the validation of the timestamp values in the message received over the channel. 'vSIGN' represents the verification of the signature information on the receiver service end. 'RecvChal' denotes the receiver process updating its receiver challenge variables. 'TS' stands for the creation of a timestamp values for the message to be send back to the sender service. 'SIGN' and 'ENCRYPT' initiates assigning the signature information to the message and encrypting the message, respectively. 'Send/Recv_Msg' illustrates the sending of the message over the channel. 'DECRYPT', 'TS', 'SIGN' represents the decryption, validating the timestamp, and the signature information at the sender service end. 'sBIND' denotes the commitment of the sender service to the receiver service. 'TS', 'SIGN', 'ENCRYPT' and 'Send/Recv_Msg' represent the allocation of the timestamp and signature information to the message, encrypting with the receiver's public key, and sending the message on the channel. 'DECRYPT', 'TS', 'vSIGN' denotes the decryption of the message by the receiver's public key, and validating the timestamp and signature information, respectively. 'rBIND' denotes the binding of the receiver to the sender service. Each time input is read, a stack function is executed and a transition is made to a new state.

The Simple Message Exchange Protocol model consists of three main steps common to service A and service B before the message is sent on the channel. Step1: the message is allocated a timestamp value, which shows

Table 4.1: Stacks for Simple Message Exchange Protocol

Stack A	**Stack B**
SenderChallenge(A,B)	decrypt(priv(B))
cTimeStamp()	vTimeStamp()
sign(A)	vSign(A)
encrypt(pub(B))	RecvrChallenge(A,B)
decrypt(priv(A))	cTimeStamp
vTimeStamp	sign(B)
vSign(B)	encrypt(pub(A))
SenderBind(A,B)	decrypt(priv(B))
cTimeStamp()	vTimeStamp()
sign(A)	vSign(A)
encrypt(pub(B))	RecvrBind(A,B))
y	z

when the message was created and when the message expires. Step2: the message is then signed by the sending service and the signature structure is embedded into the message. Step3: the message is encrypted by the receiver's public key and sent over the channel. When the message is received by the service on the other end, three steps are performed in general. Step1: the message is decrypted using the private key of the receiver service. Step2: the freshness of the message is checked by validating the timestamp. Step3: the signature is validated – it should match the signature for the sending service.

Before the protocol run is described, the stacks used in the SMEP in Table 4.1 will be discussed. An added advantage of a pushdown automaton is the infinite memory it provides in the form of a stack. This feature is beneficial in designing complex protocols where multiple goals have to be satisfied during a single protocol run. This leads to increase in functionality.

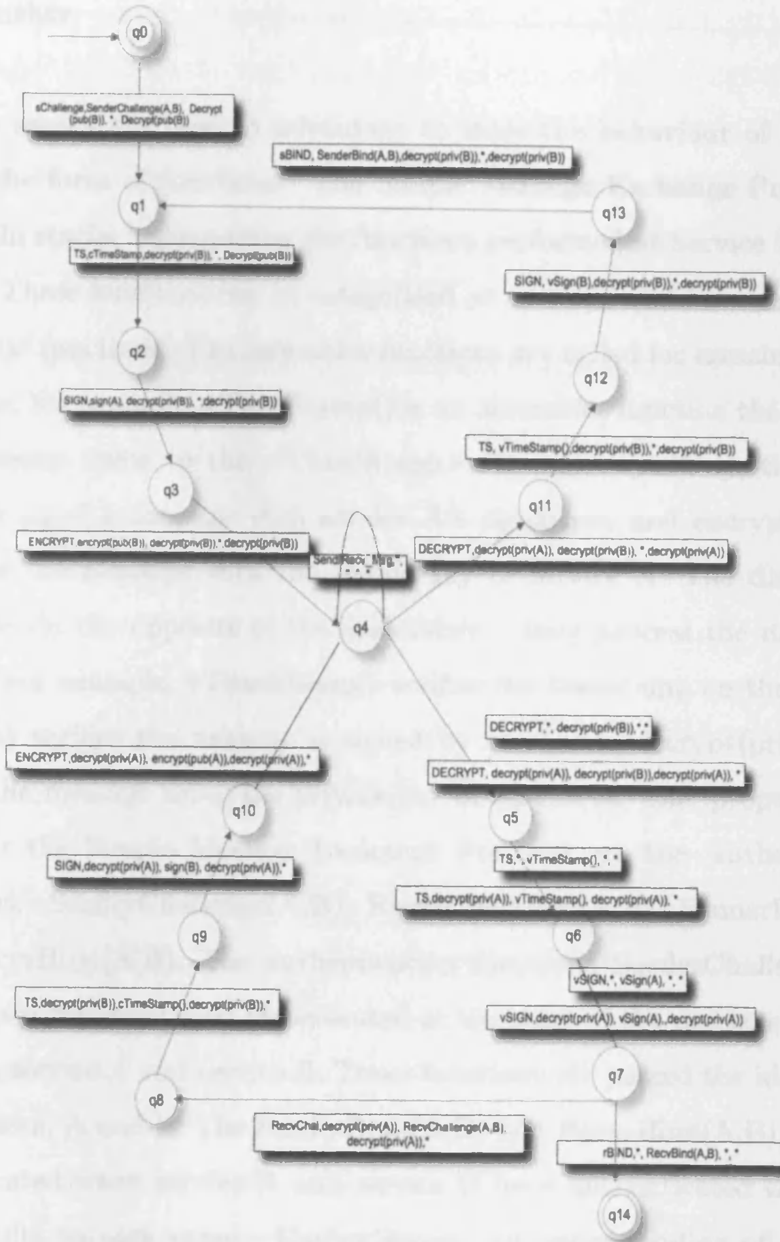
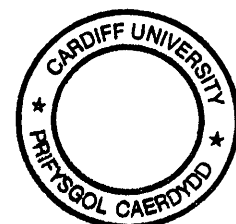


Figure 4.1: Simple Message Exchange Protocol automaton

The infinite stack memory provides the services with the ability to add more functionality.

The stacks are used to advantage to store the behaviour of the protocol in the form of functions. The Simple Message Exchange Protocol has two main stacks, representing the functions performed at Service A and Service B. These functions can be categorised as 'assembler', 'disassembler', and 'property' functions. The assembler functions are called for creating message elements, for example, cTimeStamp() is an assembler function that allocates a timestamp value to the <TimeStamp> element. Similarly, the sign(A) function signs a message with service A's signature, and encrypt(pub(A)) encrypts the message with the public key of service A. The disassembler functions do the opposite of the assemblers – they process the message received. For example, vTimeStamp() verifies the timestamp on the message, vSign(A) verifies the message is signed by service A, decrypt(priv(A)) decrypts the message using the private key of service A. The 'property' functions for the Simple Message Exchange Protocol are the 'authentication' functions: SenderChallenge(A,B), RecvrChallenge(A,B), SenderBind(A,B) and RecvrBind(A,B). The authentication functions SenderChallenge(A,B) and RecvrChallenge(A,B) are executed at the start of the message exchange between service A and service B. These functions are passed the identities of the services, A and B. The SenderBind(A,B) and RecvrBind(A,B) functions are executed when service A and service B have authenticated themselves successfully to each other. Having gained an understanding of the stack functions, the pushdown automaton model for the Simple Message Exchange Protocol will now be discussed.

Figure 4.1 defines the pushdown automaton for the Simple Message Exchange Protocol. A transition function is represented as a set of symbols involving input from the tape and functions on the stack. A transition function is denoted as {input symbol, pop first stack function, pop second stack function, push first stack function, push second stack function}. Suppose Service A wants to exchange some message with another service B. 'q0' and 'q14' are the initial and accept states of the protocol run. Service A reads the first input symbol, sChallenge, from the input tape, and functions are popped from both the stacks. The function SenderChallenge(A,B) is popped from Service A's stack. The function is executed as soon as it is popped from the stack. The function decrypt(pub(B)) is popped from Service B's stack. Nothing is pushed back onto Service A's stack. However in the case of Service B, the function is pushed back on the stack, as we do not want any action to be performed on Service B's stack while the message is being composed by service A. This transition is presented as {sChallenge, SenderChallenge(A,B), decrypt(priv(B)), *, decrypt(priv(B))}. The first element of the transition function is the input that is read, the second is the function that is popped from Service A's stack, the third element represents the function popped from Service B's stack, the fourth represents the element pushed onto service stack A ("*" means that no action is performed), the fifth element represents the function pushed onto Service B's stack. A transition from state 'q0' to state 'q1' has now been made. Next TS is read from the input tape and a timestamp is assigned to the message. The following transition function represents a transition from 'q1' to 'q2': {TS, cTimeStamp(), decrypt(pub(B)), *, decrypt(pub(B))}. cTimeStamp() is executed from Service A's stack and no action is performed on Service B's stack. For a transition to the next state 'q3', SIGN is read from the input tape and the fol-



lowing transition function is performed: {SIGN, sign(A), decrypt(priv(B)), *, decrypt(priv(B))}. sign(A) is executed from Service A's stack, and at the end of the transition Service B's stack remains unchanged. The last step before sending the message on the channel is to encrypt it. The {ENCRYPT, encrypt(pub(B)), decrypt(priv(B)), *, decrypt(priv(B))} transition is made to move to state 'q4'. The function encrypt(pub(B)) is popped and executed, resulting in the message being symbolically encrypted with Service B's public key. The message is then sent on the insecure channel. Send/Recv_Msg is read from the tape.

At Service B the message is received from the channel. At the receiver end functions from Service B's stack are executed and no action is performed on Service A's stack. DECRYPT is read from the input tape and the following transition is performed to move from state 'q4' to 'q5': {DECRYPT, decrypt(priv(A)), decrypt(priv(B)), decrypt(priv(A)), *}. If there are multiple transition functions to choose from between two states, one transition function is selected based on the values on the top of the stack. The message is decrypted by calling the 'decrypt(priv(B))' function, i.e., Service B uses its private key to perform symbolic decryption. After the message has been decrypted, TS is read to move to the next state, 'q6', and the message timestamp is checked. The {TS, decrypt(priv(A)), vTimeStamp(), decrypt(priv(A)), *} transition is executed. vTimeStamp() is popped from the stack to verify the freshness of the message. Service B reads the next element on the input tape, vSIGN, and moves to state 'q7' once the signature is verified by the following transition: {vSIGN, decrypt(priv(A)), vSign(A), decrypt(priv(A)), *}. vSign(A) is popped and executed from service B's stack – it is passed the identity of Service A to verify the signature elements

for Service A. The next alphabet symbol read from the input tape is RecvrChal to shift from state 'q7' to 'q8'. The transition function is{RecvChal, decrypt(priv(A)), RecvChallenge(A,B), decrypt(priv(A)), *}, and the effect is that Service B updates the receiver challenge variables. Service B creates a response message by executing the following sequence of events. Service B moves from state 'q8' to 'q9' by reading TS and executing { TS, decrypt(priv(A)), cTimeStamp(), decrypt(priv(A)), *}. The resulting message is assigned a timestamp value for creation and expiration by the cTimeStamp() function. SIGN is read from the tape and the state transitions from 'q9' to 'q10' after the transition { SIGN, decrypt(priv(A)), sign(B), decrypt(priv(A)), *} is performed. sign(B) is popped and executed resulting in Service B signing the message. ENCRYPT is then read and the state advances from 'q10' to 'q11', after executing the transition {ENCRYPT, decrypt(priv(A)), encrypt(pub(A)), decrypt(priv(A)), *}, resulting in encrypting the message by running encrypt(pub(A)). The message is sent on the channel.

The message is received by Service A, and it performs the following actions: decryption, validation of freshness, verification of signature, and binding to Service B. First, DECRYPT is read from the tape in order to proceed from 'q4' to 'q11', and the following transition function is executed: {DECRYPT, decrypt(priv(A)), decrypt(priv(B)), * , decrypt(priv(A))}. decrypt(priv(A)) is popped from Service A's stack and executed resulting in the message being decrypted with the private key of Service A. The next symbol read from the tape is TS and the state shifts from 'q11' to 'q12' with transition function {TS, vTimeStamp(), decrypt(priv(B)), *, decrypt(priv(B))}. vTimeStamp() is popped from Service A's stack and is run to validate the

timestamp of the message. To move from 'q12' to 'q13' SIGN is read from the input tape, and the transition function {SIGN, vSign(B), decrypt(priv(B)), *, decrypt(priv(B))} is executed. As a consequence vSign(B) is popped from Service A's stack and executed to verify the signature of Service B. sBIND is the next input symbol read to proceed from 'q13' to 'ql'. The transition function is {sBIND, SenderBind(A,B), decrypt(priv(B)), *, decrypt(priv(B))}. SenderBind(A,B) is popped and executed, and as a result Service A binds to Service B. At this stage Service A has partially authenticated to Service B, and the last step of the message exchange is the authentication by Service B. Service A sends a response to B. It assembles the message by performing transitions in the following sequence. Service A reads TS from the input tape and moves to state 'q2'. On executing {TS, cTimeStamp(), decrypt(priv(B)), *, decrypt(priv(B))}, cTimeStamp() is popped and executed from Service A's stack which results in Service A assigning a fresh timestamp to the message. Next it reads the SIGN input symbol from the tape in order to proceed to state 'q3'. On executing {SIGN, sign(A), decrypt(priv(B)), *, decrypt(priv(B))}, sign(A) is popped and executed, and consequently Service A signs the message with its signature. Service A reads ENCRYPT from the tape and moves to state 'q4'. On executing the transition function {ENCRYPT, encrypt(pub(B)), decrypt(priv(B)), *, decrypt(priv(B))}, encrypt(pub(B)) is popped from service A's stack and executed, resulting in the message being encrypted by Service B's public key. Now the stack of Service A is empty or contains the starting symbol y. All the processing steps required for Service A has been fulfilled. The message is sent on the channel for the last time.

Service B receives this last message from Service A. Before it authenti-

cates to Service A, it executes the steps: decrypting the message, validating the timestamp and signature, and binding to Service A. DECRYPT is read from the input tape to move from state 'q4' to 'q5' by executing transition {DECRYPT, *, decrypt(priv(B)), *, *}. decrypt(priv(B)) is popped from Service B's stack, and the message is decrypted with the private key of Service B. Service B reads TS from the tape to progress to state 'q6'. The transition executed is {TS, *, vTimeStamp(), *, *}. vTimeStamp() is popped from Service B's stack and run, resulting in the message freshness being verified. The next symbol on the tape is vSIGN. Before moving to state 'q7', the following transition is executed: {vSIGN, *, vSign(A), *, *}. vSign(A) is popped from the stack and run, resulting in Service A's signature being verified. The last transition from state 'q7' to 'q14' is done when rBIND is read from the tape and transition {rBIND, *, RecvrBind(A,B), *, *} is executed. RecvBind(A,B) is popped from Service B's stack and run which causes Service B to authenticate the identity of Service A. When state 'q14' is reached and both the stacks of Service A and Service B are empty, or contain the initial elements, y and z, respectively, then it is said that a successful run of the Simple Message Exchange Protocol has been performed. Table 4.2 summarizes the above. The table represents the transition function executed at each state to move to the next state.

Each function in Table 4.1 represents functionality that is to be performed on Signature, Encryption and TimeStamp elements defined in the SMEP run. The XML syntax is extracted from WS-Security and WS-Trust which is used to model SMEP and STP. The functions inTable 4.1 are applied to the XML structures described below to either verify or create them in our model. These XML elements are described below.

Table 4.2: States and Transition Functions for SMEP.

States	Transition Functions δ
q0→q1	{sChallenge, SenderChallenge(A,B), Decrypt(priv(B)), *, decrypt(priv(B))}
q1→q2	{TS, cTimeStamp(), decrypt(pub(B)), *, decrypt(pub(B))}
q2→q3	{SIGN, sign(A), decrypt(priv(B)), *, decrypt(priv(B))}
q3→q4	{ENCRYPT, encrypt(pub(B)), decrypt(priv(B)), *, decrypt(priv(B))}
q4→q4	{Send/Recv_Msg,*,*,*,*}
q4→q5	{DECRYPT, decrypt(priv(A)), decrypt(priv(B)),decrypt(priv(A)), *}
q5→q6	{TS, decrypt(priv(A)), vTimeStamp(), decrypt(priv(A)), *}
q6→q7	{vSIGN, decrypt(priv(A)), vSign(A), decrypt(priv(A)), *}
q7→q8	{RecvChal, decrypt(priv(A)), RecvChallenge(A,B), decrypt(priv(A)), *}
q8→q9	{TS, decrypt(priv(A)), cTimeStamp(), decrypt(priv(A)), *}
q9→q10	{SIGN, decrypt(priv(A)), sign(B), decrypt(priv(A)), *}
q10→q4	{ENCRYPT, encrypt(pub(B)), decrypt(priv(B)), *, decrypt(priv(B))}
q4→q4	{Send/Recv_Msg,*,*,*,*}
q4→q11	{DECRYPT, decrypt(priv(A)), decrypt(priv(B)), *, decrypt(priv(A))}
q11→q12	{TS, vTimeStamp(), decrypt(priv(B)), *, decrypt(priv(B))}
q12→q13	{SIGN, vSign(B), decrypt(priv(B)), *, decrypt(priv(B))}
q13→q1	{sBIND, SenderBInd(A,B), decrypt(priv(B)), *, decrypt(priv(B))}
q1→q2	{TS, cTimeStamp(), decrypt(pub(B)), *, decrypt(pub(B)}
q2→q3	{SIGN, sign(A), decrypt(priv(B)), *, decrypt(priv(B))}
q3→q4	{ENCRYPT, encrypt(pub(B)), decrypt(priv(B)), *, decrypt(priv(B))}
q4→q4	{Send/Recv_Msg,*,*,*,*}
q4→q5	{DECRYPT, *, decrypt(priv(B)), *, *}
q5→q6	{TS, *, vTimeStamp(), *, *}
q6→q7	{vSIGN, *, vSign(A), *, *}
q7→q14	{rBIND, *, RecvrBind(A,B), *, *}

```
<wsu:Timestamp>
    <wsu:Created>createdStamp</wsu:Createt>
    <wsu:Expires>expiryStamp</wsu:Expires>
</wsu:Timestamp>

<wsse:UsernameToken>
    <wsse:Username> usernameValue </wsse:Username>
    <wsse:Password> passwordDigestValue </wsse:Password>
    <wsse:Nonce>nonce</wsse:Nonce>
    <wsu:Created>ustCreatedStamp</Created>
</wsse:UsernameToken>

<ds:Signature>
  <ds:SignedInfo>
    <ds:CanonicalizationMethod>... </ds:CanonicalizationMethod>
    <ds:SignatureMethod>... </ds:SignatureMethod>
    <ds:Reference>...</ds:Reference>
  </ds:SignedInfo>
  <ds:SignatureValue>signatureValue</ds:SignatureValue>
  <ds:KeyInfo>... </ds:KeyInfo>
</ds:Signature>
```

The above elements are modelled in the Promela language. Each message structure is defined by using the Promela 'typedef' construct, which is used for defining complex data types. The stack functions for Service A and Service B are executed on these structures. The Promela models are discussed in Chapter 5.

```
<xenc:EncryptedKey>
```

```
        <xenc:EncryptionMethod>...</xenc:EncryptionMethod>
        <ds:KeyInfo>
            <KeyValue>....<KeyValue>
        </ds:KeyInfo>
        <xenc:CipherData>
            <xenc:CipherValue>...</xenc:CipherValue>
        </xenc:CipherData>
        <xenc:ReferenceList>
            <xenc:DataReference>...</xenc:DataReference>
        </xenc:ReferenceList>
        <xenc:EncryptionProperties>...<xenc:EncryptionProperties>
</xenc:EncryptedKey>


<xenc:EncryptedData>
        <xenc:CipherData>
            <xenc:CipherValue>...</xenc:CipherValue>
        </xenc:CipherData>
</xenc:EncryptedData>
```

4.4 Security Token Protocol

The intended purpose of the Security Token Protocol, defined in Section 3.4, is to issue a security context. Service A sends a request for issuance of a security token to a Security Token Service (STS). The Security Token Service, after processing the request, sends a request security token response message to Service A. The response message contains a security token. More details can be read in Chapter 3. The Security Token Protocol behaviour and its goals are modelled with a pushdown automaton.

The protocol consists of the following main sequence of steps. At Service A, a request for the security token is generated. The request for security token message is timestamped, and the request security token message is signed by Service A. The request security token message is encrypted with the public key of the Security Token Service and is sent to the Security Token Service over an insecure channel. The request security token message is received by the Security Token Service which decrypts the message with its private key, validates the timestamp, and verifies whether the signature of the request message that of service A. The STS then creates a response to security token request message containing the security context token. STS sends the response to security token request message back to Service A, which processes it and accepts the security token. Service A then sends the security context token's unique identity back to the STS. The STS receives the message containing the security context identity and checks the unique identity to see if the security context identity is the same as that sent in the response to security token request message to Service A.

A pushdown automaton is comprised of three main components: the input tape, the stack, and the automaton. The input tape representing the steps involved in a complete Security Token Protocol run, from initiating a request for a security token to the acceptance of the security context between Service A and the Security Token Service, consists of following input symbols: {sChallenge, RST, TS, SIGN, ENCRYPT, Send/Recv_Msg, DECRYPT, TS, vSIGN, RecvChal, RSTR, TS, SIGN, ENCRYPT, Send/Recv_Msg, DECRYPT, TS, SIGN, sBIND, SC, TS, SIGN, ENCRYPT, DECRYPT, TS, vSIGN, rBIND, SCTID}. As the input symbols are read, some function is performed on the stack and a transition is made from one state to another, until the

final state is reached or the tape is empty. 'sChallenge' results in Service A updating the sender challenge global variables. 'RST' represents the creation of the request for a security token. 'TS', 'SIGN' and 'ENCRYPT' represents assigning a timestamp to the message, signing the message with Service A's signature, and encrypting the message with the public key of the Security Token Service, respectively. 'Send/Recv_Msg' denotes sending or receiving the message. 'DECRYPT', 'TS', 'vSIGN' and 'RecvChallenge' denote decrypting the request for security token with the private key of the Security Token Service, validating the timestamp of the request message, verifying the signature of Service A, and updating the receiver challenge variables at the Security Token Service, respectively. 'RSTR' illustrates the creation of a response to security token request message at the Security Token Service, containing security context information. 'TS', 'SIGN', 'ENCRYPT' and 'Send/Recv_Msg' represent assigning a timestamp to the response to security token request message, signing the message with the Security Token Service signature, encrypting the message with Service A's public key, and sending the response to security token request message to Service A, respectively. 'DECRYPT, 'TS','SIGN' and 'sBIND' denote decryption of the response to security token request message with the private key of Service A, validating the timestamp of the received message, verifying the Security Token Service signature on the response to security token request message, and binding to the Security Token Service, respectively. 'SC' represents the agreement of Service A on the security context token sent by the Security Token Service. 'TS','SIGN' and 'ENCRYPT represent assigning a timestamp to the new reply message containing the security context token's unique identity, signing the message with Service A's signature, and encrypting it with the Security Token Service's public key. 'DECRYPT', 'TS','vSIGN' and 'rBIND' repre-

sent the steps at the Security Token Service. These steps decrypt the message with the private key of the Security Token Service, validate the timestamp of the reply message, verify the signature of Service A, and bind to Service A. 'SCTID' denotes the Security Token Service acknowledging the security token unique identity that the Security Token Service previously sent to Service A.

The stack is another part of a pushdown automaton. There are two stacks which are used in the Security Token Protocol model presented in Table 4.3. Each service (Service A and the Security Token Service) is allocated its own stack. These stacks contain the functions that are executed when the input tape is read. The functions can be divided into three main categories: 'assemblers', 'disassemblers' and 'property' functions. The assemblers are used when constructing a message and disassemblers are used when processing the message at the receiver service. The 'property' functions validate the properties of the Security Token Protocol. The assembler functions consist of cTimeStamp(), which is used for creating a timestamp for a message. The function sign(X) is used for signing a message – it is passed the signature values of Service A or the Security Token Service. The function encrypt(pub(X)) represents the encryption of the message with the public key of the receiver service. The disassemblers are used when processing the message. vTimeStamp() is used for validating the freshness of a message, and vSign(X) is used for verifying the signature of a message. vSign(X) takes as a parameter 'X', the identity of the service (either Service A or the Security Token Service) whose signature is to be verified. The function decrypt(priv(X)) is used for decrypting the message received – the message is decrypted with the private key of the receiver service, X. The 'property functions' consist of the au-

Table 4.3: Stacks for Security Token Protocol

Stack A	**Stack STS**
SenderChallenge(A,STS)	decrypt(priv(B))
partialSC()	vTimeStamp()
cTimeStamp()	vSign(A)
sign(A)	RecvrChallenge(A, STS)
encrypt(pub(STS))	partialSC()
decrypt(priv(A))	security_context
vTimeStamp()	cTimeStamp
vSign(STS)	sign(STS))
SenderBind(A,STS)	encrypt(pub(A))
security_context()	decrypt(priv(STS))
SCTID()	vTimeStamp()
cTimeStamp()	vSign(A)
sign(A)	RecvrBInd(A,STS)
encrypt(pub(STS))	SCTID()
y	z

thentication functions: SenderChallenge(A,STS), RecvrChallenge(A, STS), RecvrBind(A,STS) and SenderBind(A, STS), which have been discussed in the previous section. In addition to the authentication functions, the property functions also contain functions related to processing security contexts. The functions partialSC(), security_context() and SCTID() are the security-context functions. The partialSC() function is executed to create a request security token and to verify a partial security context. The security_context() function is used for generating a security context after the Security Token Service agrees on the partial context. The SCTID() function is used to extract the unique security context identity from the security context received and to validate it against the one sent in the response to security token request message.

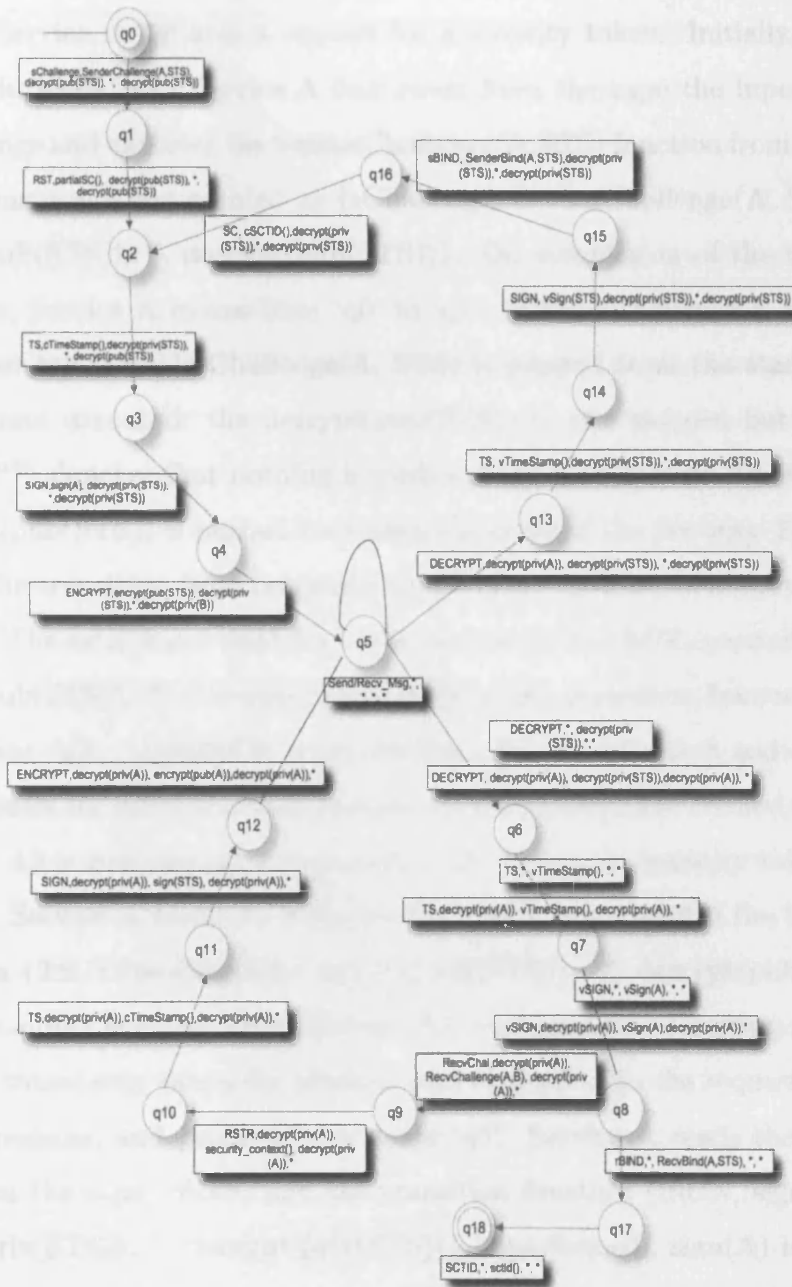


Figure 4.2: Security Token Protocol Protocol automaton.

Figure 4.2 describes the automaton model for the Security Token Protocol. Service A initiates a request for a security token. Initially, the service is in state 'q0'. Service A first reads from the tape the input symbol sChallenge and executes the SenderChallenge(A,STS) function from its stack. The transition is represented as {sChallenge, SenderChallenge(A, STS), decrypt(pub(STS)), *, decrypt(pub(STS))}. On completion of the transition function, Service A moves from 'q0' to 'q1'. When sChallenge is read from the input tape, SenderChallenge(A, STS) is popped from the stack of Service A and executed; the decrypt(pub(STS)) is also popped but not executed. "*" denotes that nothing is pushed back onto Service A's stack, and decrypt(pub(STS)) is pushed back onto the stack of the Security Token Service. The transition from one state to the other is made by observing both stacks. The next input read from the tape is RST. {RST, partialSC(), decrypt(pub(STS)), *, decrypt(pub(STS))} is the transition function to the next state, 'q2'. paritalSC() is popped from Service A's stack and executed. The request for security token elements of the message are created and populated. After completing the generation of the request security token information, Service A reads TS from the tape, and then executes the transition function {TS, cTimeStamp(), decrypt(pub(STS)) , *, decrypt(pub(STS))}. cTimeStamp() is popped from Service A's stack and run, resulting in assigning the timestamp values for creation and expiration to the request security token message, and proceeding to state 'q3'. Service A reads the next input from the tape, SIGN, and the transition function {SIGN, sign(A), decrypt(priv(STS)), *, decrypt(priv(STS))} is performed. sign(A) is popped from the stack and run, resulting in signing the request message with Service A's signature and progressing to state 'q4'. The last step before sending the request to the Security Token Service is to encrypt the message with the

public key of the Security Token Service. This is accomplished by reading ENCRYPT from the input tape, resulting in the execution of the transition function {ENCRYPT, encrypt(pub(STS)), decrypt(priv(STS)), *, decrypt(priv(STS))}. encrypt(pub(STS)) is popped from Service A's stack and executed, which results in a move to state 'q5' on completion. The message is now ready to be sent over the channel by the following transition function: {Send/Recv_Msg, *, *, *, *}.

The request security token message from Service A is now received by the Security Token Service. The Security Token Service processes the request to issue a security token and sends a response to security token request message back to Service A. The request security token message is decrypted first. DECRYPT is read from the input tape by the Security Token Service and the transition function {DECRYPT, decrypt(priv(A)), decrypt(priv(STS)), decrypt(priv(A)), *} is performed. decrypt(priv(STS)) is popped from stack of the STS and executed. The STS decrypts the message using its private key. As a result of the transition a state change occurs to 'q6'. The STS reads TS from the tape input. Here the service has the option to execute two transition functions, but only the one where the functions on the top of the stack match will be executed. The transition function {TS, decrypt(priv(A)), vTimeStamp(), decrypt(priv(A)), *} is run. The vTimeStamp() function is popped from the STS stack and run. The service verifies the timestamp values of the request security token message, and continues to state 'q7'. The STS verifies the signature of Service A on the message for a request security token, reads vSIGN from the tape and executes the transition function {vSIGN, decrypt(priv(A)), vSign(A), decrypt(priv(A)), *}. vSign(A) is popped from stack of the STS and executed. The signature

of Service A is validated, resulting in a shift to the next state, 'q8'. The STS updates its receiver challenge variables. It reads RecvChal from the input tape and runs transition function {RecvChal, decrypt(priv(A)), RecvChallenge(A,STS),decrypt(priv(A)),*}. RecvChallenge(A,STS) is popped from the STS stack and executed. The function updates the challenge variables for the Security Token Service, and as a result the service progresses to state 'q9'. The STS creates a response to security token request message for Service A containing the security context information, it reads RSTR from the tape and executes the following transition {RSTR, decrypt(priv(A)), security_context(), decrypt(priv(A)), *}. security_context() is popped from the STS stack and executed causing the request security token response information to be populated. The STS progresses to state 'q10'. The STS reads TS from the input tape, and the transition function {TS, decrypt(priv(A)),cTimeStamp(), decrypt(priv(A)), *} is performed. cTimeStamp() is popped from the STS stack and run, and the function allocates the timestamp values to the response to security token request message, and results in a state change to 'q11'. SIGN is read from the input tape, and the transition function {SIGN, decrypt(priv(A)), sign(STS), decrypt(priv(A)), *} is performed. sign(STS) is popped from the STS stack and executed, which allows the request security token response message to be signed by the Security Token Service. At the end of the transition progress is made to state 'q12'. The last symbol to be read from the tape by the Security Token Service before sending the request security token response message to service A is ENCRYPT. This leads to the transition function {ENCRYPT, decrypt(priv(A)), encrypt(pub(A)), decrypt(priv(A)), *} being performed. encrypt(pub(A)) is popped from the stack and run. The function symbolically encrypts the response to security token request message with the public key of A, thus causing a state change

to state 'q5'.

The response to security token request message is received by Service A, and is first decrypted. DECRYPT is read from the input tape, and the transition function {DECRYPT, decrypt(priv(A)), decrypt(priv(STS)), *, decrypt(priv(STS))} is performed. The decrypt(priv(A)) function is executed, and the message is symbolically decrypted using the private key of Service A. decrypt(priv(STS)) is pushed back on the stack of the Security Token Service and the state is progressed to state 'q13'. No actions are performed on the STS stack, and the function popped is pushed back on to the stack without being executed. Service A reads TS from the tape, and transition function {TS, vTimeStamp(), decrypt(priv(STS)), *, decrypt(priv(STS))} is performed. vTimeStamp() is popped from Service A's stack and executed, and the state is changed to 'q14'. SIGN is read from the input tape. The read initiates transition function {SIGN, vSign(STS), decrypt(priv(STS)), *, decrypt(priv(STS))}. vSign(STS) is popped from Service A's stack and run. The function verifies the signature of the Security Token Service. The transition causes the state to be modified to 'q15'. Service A reads the next symbol on the tape, sBIND, and this leads to the the transition function {sBIND, SenderBind(A,STS), decrypt(priv(STS)), *, decrypt(priv(STS))} being performed. SenderBind(A,STS) is popped from Service A's stack and executed. The function updates the sender bind variables on Service A's side. After the completion of the transition the state is changed to 'q16'. The last part of the Security Token Protocol involves Service A sending the unique identity of the security context back to the STS. This is done to let the STS know that the security context received by Service A was the same as that sent by the Security Token Service. A message is created containing the security context identity

sent by the STS, SC is read from the input tape and the transition function {SC, cSCTID(), decrypt(priv(STS)), *, decrypt(priv(STS))} is performed. The security context identity is extracted and the state is adjusted to the next state in the system, 'q2'. The message is then timestamped, signed by Service A and encrypted with the public key of the STS before being sent on the channel. The following transitions and state changes are made respectively. {TS, vTimeStamp(), decrypt(priv(STS)), *, decrypt(priv(STS))}: state change to 'q3'. {SIGN, sign(A), decrypt(priv(STS)), *, decrypt(priv(STS))}: state change to 'q4'. {ENCRYPT, encrypt(pub(STS)), decrypt(priv(STS)), *, decrypt(priv(STS))}: state change to 'q5'. The message is sent to the STS.

The message is then received by the STS. The service reads DECRYPT from the input tape, causing the transition function {DECRYPT, *, decrypt(priv(STS)), *, *} to be performed. decrypt(priv(STS)) is popped from the STS stack and run. The function symbolically decrypts the message with its private key. At the end of the transition progress is made to state 'q6'. TS is read from the input tape, resulting in transition function {TS, *, vTimeStamp(), *, *} being performed. vTimeStamp() is popped from the STS stack, the freshness of the message is validated, and the state is adjusted to 'q7'. STS reads vSIGN from the tape, and as a consequence the transition function {vSIGN, *, vSign(A), *, *} is performed. vSign(A) is popped from the stack and run, and the signature of Service A on the message is validated and the state changes to 'q8'. The service reads rBIND next, causing the transition function {rBIND, *, RecvrBind(A,STS),*, *} to be performed. RecvrBind(A,STS) is popped from the STS stack and executed, the function updates the receiver bind variables at the STS, and as a result of the transition the state is changed to 'q17'. The last symbol on the input tape SC

is read by the Security Token Service, and as a consequence the transition function {SCTID(), *, sctid(), *, *} is performed. sctid() is popped from the STS stack and run. This function validates the unique identifier for the security context, and the final state 'q18' is reached. This concludes a single complete run of the Security Token Protocol. The above transitions and state changes are summarized in Table 4.4.

Each function in Table 4.3 represents functionality that is to be performed on Signature, Encryption, TimeStamp , Request Security Token, and Response to Security Token Request XML elements defined in the Security Token Protocol run. These structures are extracted from the WS-Trust standard. The functions defined in table 4.3 are applied to these XML structures during each step of the protocol run.The TimeStamp, Signature, Encryption structure has been described in Section 4.3.

The Request Security Token structure is embedded in the WS-Security structure when requesting a security token. Below is a structure for requesting a STS to issue a security token used in our Security Token Service Protocol. <TokenType> represents the type of token being requested, represented as a URI. <RequestType> defines a URI that represents the function being requested. <AppliesTo> defines a scope for which the token is valid.

```
<wst:RequestSecurityToken>
    <wst:TokenType>...</wst:TokenType>
    <wst:RequestType>...</wst:RequestType>
    <wsp:AppliesTo>...</wsp:AppliesTo>
    <wst:Entropy>
        <wst:BinarySecret>...</wst:BinarySecret>
```

Table 4.4: States and Transition Functions for STP.

States	Transition Functions δ
q0→q1	{sChallenge, SenderChallenge(A, STS), decrypt(pub(STS)), *, decrypt(pub(STS))}
q1→q2	{RST, partialSC(), decrypt(pub(STS)), *, decrypt(pub(STS))}
q2→q3	{TS, cTimeStam(), decrypt(pub(STS)) , *, decrypt(pub(STS))}
q3→q4	{SIGN, sign(A), decrypt(priv(STS)), *, decrypt(priv(STS))}
q4→q5	{ENCRYPT, encrypt(pub(STS)), decrypt(priv(STS)), *. decrypt(priv(STS))}
q5→q5	{Send/Recv_Msg, *, * , *, *}
q5→q6	{DECRYPT, decrypt(priv(A)), decrypt(priv(STS)), decrypt(priv(A)), *}
q6→q6	{TS, decrypt(priv(A)), vTimeStamp(), decrypt(priv(A)), *}
q7→q8	{vSIGN, decrypt(priv(A)), vSign(A), decrypt(priv(A)), *}
q8→q9	{RecvChal, decrypt(priv(A)), RecvChallenge(A,STS),decrypt(priv(A)),*}
q9→q10	{RSTR, decrypt(priv(A)), security_context(), decrypt(priv(A)), *}
q10→q11	{TS, decrypt(priv(A)),cTimeStamp(), decrypt(priv(A)), *}
q11→q12	{SIGN, decrypt(priv(A)), sign(STS), decrypt(priv(A)), *}
q12→q5	{ENCRYPT, decrypt(priv(A)), encrypt(pub(A)), decrypt(priv(A)), *}
q5→q5	{Send/Recv_Msg, *, * , *, *}
q5→q13	{DECRYPT, decrypt(priv(A)), decrypt(priv(STS)), *, decrypt(priv(STS))}
q13→q14	{TS, vTimeStamp(), decrypt(priv(STS)), *, decrypt(priv(STS))}
q14→q15	{SIGN, vSign(STS), decrypt(priv(STS)), *, decrypt(priv(STS))}
q15→q16	{sBIND, SenderBind(A,STS), decrypt(priv(STS)), *, decrypt(priv(STS))}
q16→q2	{SC, cSCTID(), decrypt(priv(STS)), *, decrypt(priv(STS))}
q2→q3	{TS, cTimeStam(), decrypt(pub(STS)) , *, decrypt(pub(STS))}
q3→q4	{SIGN, sign(A), decrypt(priv(STS)), *, decrypt(priv(STS))}
q4→q5	{ENCRYPT, encrypt(pub(STS)), decrypt(priv(STS)), *. decrypt(priv(STS))}
q5→q5	{Send/Recv_Msg, *, * , *, *}
q5→q6	{DECRYPT, *, decrypt(priv(STS)), *, *}
q6→q7	{TS, *, vTImeStamp(), *, *}
q7→q8	{vSIGN, *, vSign(A), *, *}
q8→q17	{rBIND, *, RecvrBind(A,STS),*, *}
q17→q18	{SCTID(), *, sctid(), *, *}

```
    </wst:Entropy>
    <wst:Lifetime>
        <wsu:Created>...</wsu:Created>
        <wsu:Expires>...</wsu:Expires>
    </wst:Lifetime>
</wst:RequestSecurityToken>
```

Entropy defines the value to be used in creating a key. The value should be <xenc:EncryptedKey> or <wst:BinarySecret>. The BinarySecret element specifies a base64 encoded sequence of octets representing the requestor's entropy. The <wst:Lifetime> is similar to the <wst:TimeStamp> reflecting the same information when the token was created and when it expires. They are used interchangeably while designing protocols.

A structure for a response from a STS for the request to issue a security token will now be given. The response should contain all the elements originally present in the request except the requestor's entropy and the token requested. The keys are calculated using partial entropy, i.e., both the Security Token Service and the requestor provide entropies. This entropy is used to calculate the keys. When keys resulting from a token request are not directly returned and must be computed, the computed keys are represented using a <wst:ComputedKey> element placed inside a <wst:RequestedProofToken>. The <wst:ComputedKey> element returns how the key is to be computed, e.g., SHA-1. The value of the computed key returned is in the form of a URI.

```
<wst:RequestSecurityTokenResponse>
    <wst:TokenType>...</wst:TokenType>
    <wsp:AppliesTo>...</wsp:AppliesTo>
```

```
    <wst:RequestType>...</wst:RequestType>
    <wst:RequestedProofToken>
        <wst:ComputedKey>...</wst:ComputedKey>
    </wst:RequestedProofToken>
    <wst:Entropy>
        <wst:BinarySecret>...</wst:BinarySecret>
    </wst:Entropy>
    <wst:Lifetime>
        <wsu:Created>...</wsu:Created>
        <wsu:Expires>...</wsu:Expires>
    </wst:Lifetime>
</wst:RequestSecurityTokenResponse>
```

4.5 Concluding Remarks

This chapter has presented an automaton model of the Simple Message Exchange Protocol and the Security Token Protocol, described in Chapter 3. The automaton model describes the behaviour of the protocols in an explicit way. The next step is the verification of the model using Spin. The protocols will be expressed in the Promela language, and Service A, Service B and the Service Token Service will be defined as the participants of the protocols.

Other automaton based work includes Diaz [DPC+06] who has modelled web services into timed automata by applying time restrictions and Fu [Fea04] use guarded finite state automata to represent web services. Both focuses on composite web services while our work focus on protocols for standard web services and the finding of new approach that can combine both behaviour and the properties of a protocol in a single model.

Increasing the number of requesting services in STP run might degrade the performance causing delayed responses or packet loss. Each requesting party can be allocated a separate thread to process the request, thus allowing the security token service to continue with the other pending requests.

SMEP and STP are mainly designed for two services. In the situation where multiple services are involved, there will be no extra impact on the stacks as each service has its own stack. In the case of SMEP, multiple services involved in the protocol run can be classified as sending and receiving services.

CHAPTER 5

Promela Model

A system is correct if it meets its design requirements. However, this statement is not enough to prove the correctness of concurrent systems. The real test for these systems is that they do not fail to meet the specified requirements.

General-purpose model checking tools can be used for verification of a system model, or tools can be built that are targeted for the verification of the particular model, such as Tulafale [BF04]. This thesis uses *Spin*, a general-purpose model checker, for verification of the model. The specification language accepted by *Spin* is Promela, which allows the facts about a transition system to be modelled while hiding the lower-level details about the system. *Spin* is a state exploration model checker and has algorithms for state space reduction, thus controlling the size of the state space. *Spin* is

used to verify the correctness requirements for the concurrent system. There are two ways *Spin* can be used for model checking: *(i)* using *Spin* to construct the verification model for the system at hand, or *(ii)* starting from the implementation and converting critical parts of the implementation into a verification model which is then analysed using *Spin*.

The modelling language for *Spin* is called Promela. In Promela it is very difficult to specify any computations that are not rudimentary. This allows for specifying the infrastructure and mutual dependency of concurrently executing processes.

5.1 Chapter Objectives

This chapter presents the translation of the system model presented in Chapter 4 for the Simple Message Exchange Protocol and the Security Token Protocol, and their Linear Temporal Logic properties, into the Promela language. The basic components that form the Promela model are defined for the Simple Message Exchange Protocol and Security Token Protocol. The purpose of the model checking exercise is not to build and analyse verification models that are as detailed as possible, but to find and build the smallest sufficient model to describe the essential elements of the system design.

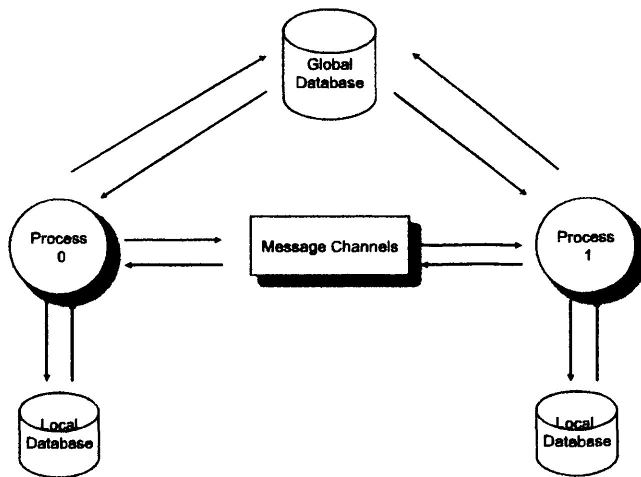


Figure 5.1: Types of Objects

5.2 Introduction to Promela

Promela is used for modelling systems for purposes of analysis and verification. Figure 5.1 presents the type of objects used when modelling a system with Promela. Global variables, which are accessible to all the processes in the system, may be defined, and their values can be updated by any process in the model. Processes define the functionality of the system, and have a database of local variables, only available locally. These variables are defined within the process to limit their scope. The processes can communicate with each other using message channels. The set of basic statements in Promela is small and consists of six elements: *(1)* assignments, *(2)* assertions, *(3)* print statements *(4,5)* send or receive statements, and *(6)* Promela's expression statement.

Table 5.1 presents a summary of the Promela language constructs used in the model for the Simple Message Exchange Protocol and the Security Token Protocol. The 'global database' in Figure 5.1 is composed of constants, variable declarations, mtypes, typedefs, etc., and these constructs are accessible to all the processes in the system.

A 'Process' can be defined using a 'proctype' declaration, which defines the behaviour of an executing process. They can be composed of keywords, constants, atomic statements, selection statements, channels, etc. The syntax of Promela is similar to the C language, with keywords representing some defined functionality, such as if statements, loops, etc. Constants can be defined. A constant is a sequence representing a decimal integer. Macros, or mtypes, can also be used for defining constants. Promela supports unary and binary expressions, and comparators such as $>$, $>=$, $==$, $!=$, etc. These

operators are used in defining conditional expressions, in 'if' statements and in loop constructs. The 'init' process is the instantiation process. It is the first process to be executed in a Promela model, and is used for preparing an initial state for the system.

Processes communicate with each other over message channels. There are two types of channel in Promela: buffered and rendezvous channels. Buffered channels have the ability to store messages, unlike rendezvous channels. A channel can be defined as 'chan send = [0] of {mtype}'. A message can be sent on the channel as follows, 'send ! x', and is received on a channel with 'send ? x'. A channel can be defined globally or in a process declaration. More information on Promela can be found in [Ger].

5.3 System Modelling Steps

Promela models of the Simple Message Exchange Protocol and the Security Token Protocol have been described previously in Chapter 4, which also describes the steps in modelling the protocols. The Simple Message Exchange Protocol is a 'three step' message exchange model. Service A sends a first message to Service B. The functions for encryption, decryption and assigning timestamps are applied to the message before sending it off on the channel. The second message is received at Service A from Service B. Service B decrypts and validates the message, and then creates and sends a third message on the channel. Each step of the protocol is modelled for Service A and Service B as atomic transactions, and each atomic transactions contains the functions applied to the messages at each state. Service

Table 5.1: Promela Constructs

Keywords	atomic, bool, typedef, do, od, if, fi, mtype, proctype, skip, else chan, bit
Constant	Can be defined as macro, or using keyword mtype, e.g. #define MAX 5
Expressions	Unary and Binary Operators or/and functions can be used to build expressions
Conditional Expressions	(exprl $\rightarrow$ expr2 : expr3)
Declarations	Processes, channels, variables must be declared before being used.
Variables	bit, bool, byte, short or int
Arrays	The same concept as in the C language
Symbolic Constants	Declared using mtype, e.g., mtype = {OK, READY, ACK}
Message Channels	Channels are used for communicating between processes, e.g., chan send = [o] of mtype
Channel Operators	Send and receive, e.g., send ! x; and send ? x; respectively
Structures	User defined data types, e.g., typedef msg{byte a; bool x;} msg m1[2];
Processes	Defined using key word proctype, e.g, proctype service(){statements}
Init Process	Used to prepare the initial state of the system, init {statements}
atomic	Executes statement in one indivisible step, e.g., atomic {statements}
Selection	Select one amongst its options, when the first statement(guard) is true, e.g., if :: statements :: statements fi

B performs two steps. It receives the first message from Service A and processes it, then sends a second message off to the sender, Service A, and waits for the response. On receiving a valid response, it authenticates to Service A.

The Security Token Protocol is also a 'three step' message exchange protocol. The first message consists of a request generated for a security token to be issued (RST), which is sent from Service A to a Security Token Service (STS). Service A waits for a response from the Security Token Service in the form of a security context. Service A then validates the second message containing the security token by applying a combination of functions to the message. Once it has validated the message it sends a third message back to the Security Token Service. This completes the protocol run on Service A's side. On the STS side, the STS waits for the first message – the request for a security token. It then processes the message and creates a response – the response to security token request (RSTR) message. A second message then sends this off to Service A. The Security Token Service waits for a third message containing the final acknowledgement, the security context identity, which was originally sent to Sservice A in the request security token response message. The protocol run completes on the STS side after the Security Token Service validates the response. These steps are modelled as 'atomic' steps. Each atomic step consists of a set of functions which are applied to the message at each state. The detailed model of these protocols will be presented in the next two subsections.

The system is modelled by breaking it into the following parts. First the 'types' used in the model are defined. These can be constants, datatypes, etc. Then the 'channel' used for communication between processes is defined.

The next step is defining the 'global variables' used in the Linear Temporal Logic Formula and the protocol environment. The 'processes' for the principals or services involved in the protocol environment are then presented. For each step a description of the protocol rules and of the protocol instance, and a description of the intruder behaviour, is provided.

5.3.1 Simple Message Exchange Protocol

The Simple Message Exchange Protocol is based on WS-Security for the exchange of messages between two services. The main goal of the protocol is authentication and secrecy between the participating principals. The Simple Message Exchange Protocol has been described in detail in Chapter 3.

Types

The constants used by the Simple Message Exchange Protocol model will now be defined. These are divided into 'general-purpose' constants, 'WS-Security' constants, 'XML Signature' constants, 'XML Encryption' constants, and 'TimeStamp' constants. The constants used in the Simple Message Exchange Protocol are presented in Table 5.2. The general-purpose constants include the participating services 'A' and 'B', and an intruder service 'I'. 'A', 'B' and 'I' represent the username, or identity, of these services. 'REQ', 'ACK', 'ACCEPT', 'DECLINE' are used in the messages to identify the message exchange state in the protocol run. The nonces, 'nonceA', 'nonceB' and 'nonceI' represent the freshly generated nonce values for Service A, Service B and the intruder. The constants used for symbolic cryptographic representation for XML Signature and XML Encryption will now be presented.

Table 5.2: Simple Message Exchange Protocol Types.

General Purpose	A,B,I,REQ,ACK,nonceA,nonceB,nonceI, ACCEPT,DECLINE
XML Signature	c14n, sha1, sigvalA, sigvalB, sigvalI, X509v3
XML Encryption	tripleDES, CD, base64encoded, RL
WS-Security	pubKeyA, pubKeyB, pubKeyI, privKeyA, privKeyB, privKeyI
TimeStamp	CREATED, EXPIRES

Constants 'c14n', 'sha1', 'X509v3' are symbolic cryptographic representations for XML Signature, and constants 'tripleDES', 'base64encoded', 'CD', etc., are symbolic cryptographic representations for XML Encryption. The 'WS-Security' constants include public and private key pairs for each service A, B and intruder I. The public keys are known to all the services in the model, but the private keys are kept secret. Lastly, the TimeStamp constant representations, 'CREATED' and 'EXPIRES', are the constants for creation and expiration of the message.

Channels

Channels are used by processes to communicate with each other. Service A, Service B and the intruder service use channels to send messages to each other. Two type of channel are declared for the Simple Message Exchange Security Protocol, and are classified according to the type of messages used for exchanging data. The first channel sends messages of type 'Msg1': {Sender, Sender_Nonce, Receiver, REQ, TimeStamp, SignatureInfo, EncryptedData, pubKey(Receiver)}. This message consists of the username of the sender, the nonce of the sender, the username of the receiver to identify the receiver process at both sides of the message exchange, the status of the message,

e.g., REQ, ACK, ACCEPT or DECLINE, the timestamp associated with the message, the signature information, the encrypted data information, and the public key of the receiver process, which indicates that the message has been encrypted by the sender service using the receiver's public key. The second channel is used to pass messages of type 'Msg2': {Sender, Sender_Nonce, Receiver, Receiver_Nonce, RES/ACCEPT, TimeStamp, SignatureInfo, EncryptedData, pubKey(Receiver)}. The message contains the sender's name and nonce, the receiver's name and nonce, the message status, 'TimeStamp', 'SignatureInfo' and 'EncryptedData', and the public key representing the symbolic encryption done using the public key of the receiver.

A non-buffered rendezvous channel is defined for each message type. chanONE is used for sending message 'Msg1', and chanTwo is used for sending the message 'Msg2'. The declarations are listed below:

```
chan chanONE = [0] of {mtype, mtype, mtype, mtype, TimeStamp, Signature,
EncryptedData, mtype};

chan chanTWO = [0] of {mtype, mtype, mtype, mtype, mtype, mtype, TimeStamp,
Signature, EncryptedData, mtype};
```

Global Variables

Linear Temporal Logic (LTL) is used to define the properties the system is supposed to satisfy. The Simple Message Exchange Protocol satisfies the properties of authentication and secrecy, and so these properties are modelled with LTL. Various other global variables are used to track the message exchange in the SMEP model.

Four variables are used for checking the LTL properties: '*SenderChal-*

lengeAB', '*SenderBindAB*', '*RecrChallengeAB*' and '*RecvrBindAB*'. 'SenderChallengeAB' is set to true when the sender Service A initiates a run with Service B. 'RecvrChallengeAB' is set to true when Service B knows it is talking to Service A. 'SenderBindAB' is true when Service A commits to Service B, in other words, when Service B authenticates to Service A. Similarly, 'RecvrBindAB' is true when Service B binds or commits to Service A. This model can be adopted for session key establishment. The four variables are defined as follows:

```
bit SenderChallengeAB=0;

bit SenderBindAB=0;

bit RecvrChallengeAB=0;

bit RecvrBindAB=0;
```

The SMEP properties are now mapped to the LTL formalism which is then directly input to the *Spin* model checker.

In the LTL property below, '[]' represents $\Box$, meaning "always". '!' is used for negation, '||'represents "or", and 'U' represents "until". The property reads as follows: 'SenderBindAB' is always false, or is not true until 'RecvrChallengeAB' is true. Similarly, 'RecvrBindAB' is always false, or is not true until 'SenderChallengeAB' is true.

```
[] ( ([] !SenderBindAB) || (!SenderBindAB U RecvrChallengeAB) )
[] ( ([] !RecvrBindAB) || (!RecvrBindAB U SenderChallengeAB) )
```

TimeStamp, XML Signature, XML EncryptedData and EncryptedKey are complex structures and are defined using 'typedef' structures which are

listed in Table 5.3. Each element of the typedef structure represents an element of the XML structure. The values assigned to these structures represent the contents of the elements.

Other global variables used can be found in Appendix SMEP.

Processes of Principals in the Network

The Simple Message Exchange Protocol consists of three principals or services. Service A and Service B represent the legitimate users of the system, and the intruder service represents the actions of an intruder. These services are modelled as three processes: 'SenderA', 'ReceiverB' and 'PI' (the intruder process). An 'init' process to instantiate the model is also defined.

Init Process: The 'init' process contains the instantiation statement for all three processes. It allows the sender to initiate a protocol run with either Service B or Service I, and starts the intruder process and the receiver processes. The 'init' process is described below.

```
init {
   if
   :: run SenderA(A,B,nonceA)
   :: run SenderA(A,I,nonceA)
   fi
   run ReceiverB(B,nonceB)
   run PI(I, nonceI)
}
```

Table 5.3: Simple Message Exchange Protocol Global Variables.

<table>
<tr><td>TimeStamp</td><td>typedef TimeStamp{
    mtype Created;
    mtype Expires;
}; TimeStamp ts[6];</td></tr>
<tr><td>XML Signature</td><td>typedef Signature {
    mtype CanoncalizationMethod;
    mtype SignatureMethod;
    mtype Reference;
    mtype SignatureValue;
    mtype KeyInfo;
}; Signature sig[6];</td></tr>
<tr><td>XML Encryption - Encrypted Data</td><td>typedef EncryptedData{
    mtype EncryptionMethod;
    mtype CipherData;
    mtype CipherValue;
    mtype ReferenceList;
    mtype DataReference;
    mtype KeyInfo;
}; EncryptedData edata[6];</td></tr>
<tr><td>XML Encryption - Encrypted Key</td><td>typedef EncryptedKey {
    mtype EncryptionMethod;
    mtype CipherData;
    mtype CipherValue;
    mtype ReferenceList;
    mtype DataReference;
    mtype KeyInfo;
}; EncryptedKey ek[6];</td></tr>
</table>

Sender Process: The sender process initiates communication with the receiver process 'ReceiverB' or with the intruder process 'PI'. It takes three arguments on instantiation: its identity A, the nonceA, and the process it communicates with. The sender process executes the 'SenderChallenge(X,Y)' function, and sets the 'SenderChallengeAB' to true if it is starting a run with B. Before sending the message over the channel, it populates the XML Signature, XML Encryption, and TimeStamp structures. It then sends the message on channel 'chanONE'. After that, the sender process waits for a response from the receiver process on 'chanTWO'. When it receives this message it decrypts the message with its private key and checks for the symbolic TimeStamp values, verifies the signature elements of the message, and then calls the 'SenderBind(X,Y)' function. If it is in a run with B, it will set its 'SenderBindAB' to true and sends an ACCEPT message back to the service. The following is a definition of the sender process.

```
proctype SenderA(mtype me; mtype recvr; mtype my_nonce)
{
    atomic {
        SenderChallenge(me, recvr);
        chanONE ! me,my_nonce,recvr,REQ,ts[0],sig[0],edata[0],pubKey
    }
    atomic {
        chanTWO ? recvr,recvr_nonce,me,my_nonce,ACK,ts[3],sig[3],edata[3],pubKey;
        Decryption(pubKey, privateKey);
        VerifyFreshness(ts[3].Created,ts[3].Expires);
        VerifySignature(sig[3].CanoncalizationMethod, sig[3].SignatureMethod,
            sig[3].Reference, sig[3].SignatureValue, sig[3].KeyInfo);
        SenderBind(me,recvr);
        sig[0].Reference = ACCEPT;
        chanTWO ! me,my_nonce,recvr,recvr_nonce,ACCEPT,ts[0],
```

```
        sig[0], edata[0], pubKey;
    }
}
```

'SenderChallenge(X,Y)' and 'SenderBind(X,Y)' are macros used for authentication between the sender and the receiver. These macros are called by the sending service to update the values of the LTL global variables 'SenderChallengeAB' and 'SenderBindAB'. In order to explain this part, it is necessary to explain the technique used for property specifications, which is similar to the one presented in [Low96a]. The fact that a responder with identity B correctly authenticates to an initiator with identity A can be expressed by the following proposition: A commits to a session with B only if B has indeed taken part in a run of the protocol with A. A similar proposition expresses the reciprocal property, i.e., the fact that an initiator with identity A correctly authenticates to a responder with identity B. Each one of the basic propositions involved in the above properties can be represented in Promela by means of a global boolean variable which becomes true at a particular stage of a protocol run. These macros are defined as follows:

```
#define SenderChallenge(x,y) if
    :: (x==A && y==B) → SenderChallengeAB=1
    :: (x==A && y==I) → SenderChallengeAI=1
    :: (x==I && y==B) → SenderChallengeIB=1
    :: else skip
    fi


#define SenderBind(x,y) if
```

```
:: ((x==A)&&(y==B)) → SenderBindAB=1
:: ((x==A)&&(y==I)) → SenderBindAI=1
:: ((x==i)&&(y==B)) → SenderBindIB=1
:: else skip
fi
```

Receiver Process: The receiving service B is limited to receiving messages. It cannot start a conversation but can receive and process messages from Service A and the intruder service, PI. The receiver process is modelled as follows. Firstly, Service B waits for a message on 'chanONE'. On receiving the message it symbolically decrypts the message with its private key 'privB', verifies the freshness of the message and the signature on the message, and calls the 'RecvrChallenge(X,Y)' routine to update the global variables. Then it sends a response back to the sending service. Next Service B waits for a response from the sending service on 'chanTWO', decrypts the message, verifies the freshness of the message and the signature, and finally binds to the sending service. The definition of the receiver process is listed as follows.

```
proctype ReceiverB(mtype me; mtype my_nonce){
    atomic {
        chanONE?sender,sender_nonce,eval(me),msg_type,ts[1],
            sig[1], edata[1], pubKey;
        Decryption(pubKey, privateKey);
        VerifyFreshness(ts[1].Created,ts[1].Expires);
        VerifySignature(sig[1].CanoncalizationMethod, sig[1].SignatureMethod,
            sig[1].Reference, sig[1].SignatureValue, sig[1].KeyInfo);
        RecvrChallenge(sender,me);
        chanTWO!me,my_nonce,sender,sender_nonce,ACK,ts[2],
```

```
            sig[2],edata[2],pubKey
    }
    atomic {
        chanTWO ? eval(sender),eval(sender_nonce),eval(me),
            eval(my_nonce),ACCEPT,ts[4],sig[4],edata[4],pubKey;
        Decryption(pubKey, privateKey);
        VerifyFreshness(ts[4].Created,ts[4].Expires);
        VerifySignature(sig[4].CanoncalizationMethod, sig[4].SignatureMethod,
            sig[4].Reference, sig[4].SignatureValue, sig[4].KeyInfo);
        RecvrBind(sender,me);
    }
}
```

The receiver process has its own 'RecvrChallenge(X,Y)' and 'RecvrBind(X,Y)' macros. They are similar to the ones described for the sender macros and are updated in the same manner.

PI Process: The process 'PI' is the intruder process, and is based on the Dolev-Yao model for an intruder. The intruder process has the capability to overhear and intercept any message sent on the channel between Service A and Service B, and can create new messages based on the information learned. The Dolev-Yao model assumes that cryptography cannot be broken during a protocol run. The intruder can start a conversation with either Service A or Service B. The intruder can behave like a normal service, thus allowing other participants to initiate a protocol run with it, or vice versa. The intruder can start a fresh conversation with either A or B as a valid user. By doing so, the intruder is able to learn the nonces for Service A and Service B.

The behaviour of the intruder depends on the knowledge it originally has and the knowledge it will acquire during protocol runs with the participants. For example, the intruder's knowledge might include the intruder's identity, its public and private key, the identities of other participating agents, their public keys and any other secrets shared between the intruder and participants.

There are four basic message runs the intruder can perform:

1. I $\rightarrow$B: the intruder can talk to B as a new user and learn its nonce.
2. A $\rightarrow$I: the intruder can interact with A and learn its nonce.
3. A$\rightarrow$(I)B: A can start a conversation with the intruder masquerading as Service B.
4. I(A) $\rightarrow$ B: the intruder, masquerading as Service A, can start a conversation with Service B.

These last two runs, in which the intruder behaves as either Service B or Service A can be performed only when the intruder has sufficient knowledge obtained during previous runs.

The intruder already knows the public keys of A and B. When it learns the nonces of A and B, it updates its knowledge base using the macro defined below.

```
#define k(x1) if
    :: (x1 == nonceA)→ learn_kNa = 1;
    :: (x1 == nonceB)→ learn_kNb = 1;
    :: else skip
    fi
```

The intruder process PI is an always running process. It is either listening on the channel or is sending a message on the channel.

In the first of the four cases above the intruder initiates a run with Service B, as a legitimate user, i.e., I→B. The intruder updates its global variables for SenderChallenge, populates the Signature and Encryption information, sends the message on 'chanONE' to Service B, and waits for a response. On receiving the response, it decrypts the message, verifies its freshness, validates the Signature, and learns the nonce of Service B. The intruder then sends a message back to service B. Once the intruder has learned the nonce of B it can now use this in a protocol run between Service A and itself acting as Service B. Similarly, in the second of the four cases above, the intruder can have a message run with Service A and learn its nonce. The intruder, after learning the nonce, acts as an impostor and can participate in message exchanges pretending to be Service B or Service A. A segment of the intruder model for I→B is given as follows :

```
atomic {
    SenderChallenge(me, B);
    chanONE ! me,my_nonce,B,REQ,ts[1],sig[1],edata[1],pubKeyB ;
    chanTWO?recvr,recvr_nonce,me,my_nonce,ACK,ts[5],sig[5],edata[5],pubKey;
    Decryption(pubKey, privateKey);
    VerifyFreshness(ts[5].Created,ts[5].Expires);
    VerifySignature(sig[5].CanoncalizationMethod, sig[5].SignatureMethod,
        sig[5].Reference, sig[5].SignatureValue, sig[5].KeyInfo);
    SenderBind(sender, me);
    sig[1].Reference = ACCEPT;
    chanTWO !me,my_nonce,recvr,recvr_nonce,ACCEPT,ts[1],sig[1],edata[1],pubKey;
}
```

The second of the four cases above Service A takes the intruder to be a legitimate service and initiates a run with it, i.e., A→I. The intruder interacts with Service A and learns its nonce. The following model lists the behaviour of the intruder for this scenario.

```
atomic {
    chanONE?sender,sender_nonce,eval(me),msg_type,ts[5],sig[5],edata[5],pubKey;
    Decryption(pubKey, privateKey);
    VerifyFreshness(ts[5].Created,ts[5].Expires);
    VerifySignature(sig[5].CanoncalizationMethod, sig[5].SignatureMethod, sig[5].Reference,
        sig[5].SignatureValue, sig[5].KeyInfo);
    RecvrChallenge(sender,me);
    chanTWO!me,my_nonce,sender,sender_nonce,ACK,ts[1],sig[1],edata[1],pubKeyA;
    chanTWO?eval(sender),eval(sender_nonce),eval(me),eval(my_nonce),ACCEPT,
        ts[5],sig[5],edata[5],pubKey;
    Decryption(pubKey, privateKey);
    VerifyFreshness(ts[5].Created,ts[5].Expires);
    VerifySignature(sig[5].CanoncalizationMethod, sig[5].SignatureMethod,
        sig[5].Reference, sig[5].SignatureValue, sig[5].KeyInfo);
}
```

In the third of the four cases above the intruder acts as Service B and talks to Service A. This option sequence is only executed if the intruder has already gained knowledge of the nonce of Service B. The intruder intercepts the messages from Service A to Service B. The intruder cannot decrypt the message, however, it can use all the information which is not encrypted and signed, such as the identity of the sender and the nonce. Using this information it can call the receiver challenge and receiver bind functions. It can update the global variables and attempt to bind to Service A. The following lists the intruder model for this behaviour.

```
if
   :: learn_kNb==1→atomic {
        chanONE?sender,sender_nonce ,claim_B,msg_type,ts[5],sig[5],edata[5],pubKey;
        Decryption(pubKey, privateKey);
        VerifyFreshness(ts[5].Created,ts[5].Expires);
        VerifySignature(sig[5].CanoncalizationMethod, sig[5].SignatureMethod,
             sig[5].Reference, sig[5].SignatureValue, sig[5].KeyInfo);
        if
             ::(learn_kNb == 1)→ claim_nonceB = nonceB
        fi;
        RecvrChallenge(sender,claim_B);
        RecvrBind(sender, B);
        k(sender_nonce);
        chanTWO!claim_B,claim_nonceB,sender,sender_nonce,ACK,ts[2],sig[2],
             edata[2],pubKeyA;
        chanTWO ? eval(sender),eval(sender_nonce),claim_B,
             eval(claim_nonceB),ACCEPT,ts[5],sig[5],edata[5],pubKey;
        Decryption(pubKey, privateKey);
        VerifyFreshness(ts[5].Created,ts[5].Expires);
        VerifySignature(sig[5].CanoncalizationMethod, sig[5].SignatureMethod,
             sig[5].Reference, sig[5].SignatureValue, sig[5].KeyInfo);
   }
   ::else skip
fi;
```

In the last of the four cases above the intruder pretends to be Service A and initiates a run with Service B. This scenario can only be executed once the intruder has sufficient knowledge about Service A, i.e., its nonce. It then attempts to authenticate to Service B. The following model fragment lists this functionality.

```
atomic {
```

```
    if
        :: (learn_kNa==1)
            → chanONE!A,nonceA,B,REQ,ts[0],sig[0],edata[1],pubKeyB ;
            SenderChallenge(A, B);
            chanTWO?recvr,recvr_nonce,claim_A,claim_nonceA,ACK,ts[5],sig[5],edata[5],pubKey;
            Decryption(pubKey, privateKey);
            VerifyFreshness(ts[5].Created,ts[5].Expires);
            VerifySignature(sig[5].CanoncalizationMethod, sig[5].SignatureMethod, sig[5].Reference,
                sig[5].SignatureValue, sig[5].KeyInfo);
            k(recvr_nonce);
            SenderBind(claim_A,recvr);
            chanTWO!claim_A,claim_nonceA,recvr,recvr_nonce,ACCEPT,ts[5],sig[5],edata[5],pubKey;
            VerifySignature(sig[5].CanoncalizationMethod, sig[5].SignatureMethod, sig[5].Reference,
                sig[5].SignatureValue, sig[5].KeyInfo);
        ::else skip
    fi;
}
```

5.3.2 Security Token Protocol

The Security Token Protocol is a simple three-message exchange protocol for issuing a security token between an agent and a Security Token Service. In the Security Token Protocol the participants agree on a 'Security Context'. A security token is a collection of claims made about a client, such as statements about its identity, key, etc., and a security context refers to an authenticated state and negotiated keys. A security context token is a manifestation of this concept. The constructs that make up the Promela model for the Security Token Protocol will now be defined.

Table 5.4: Security Token Protocol – Types

General-purpose	A, STS, I, nonceA, nonceS, nonceI, DENIED
XML Signature	c14n, sha1, sigvalA, sigvalSTS, sigvalI, X509v3
XML Encryption	tripleDES, CD, base64encoded, RL
WS-Security	pubKeyA, privKeyA, pubKeySTS, privKeySTS, pubKeyI, privKeyI, dsSig
WS-Trust	partialSC, SCT, ISSUE, claimsA, claimsS, RST, RSTR, SC, created, expires, client_entropy, server_entropy, partialEntropy, SCT_ID
TimeStamp	CREATED, EXPIRES

Types

Table 5.4 represents the 'mtypes' used in the Security Token Protocol. The general-purpose mtype definitions are given for constants used in the UsernameToken profile, together with other general constants used in the model. 'A' represents the username of Service A, 'STS' represents the username of the Security Token Service, and 'I' represents the intruder of the system. Nonces are listed for each service: Service A is assigned 'nonceA', STS is assigned 'nonceS', and the intruder service has its own nonce, 'nonceI'. Nonces are important for defence against replay attacks, to avoid reuse of old communication. The mtype 'DENIED' is used to indicate whether the authentication failed between the participating agents. For the constants used in XML Signature [ERS01], 'c14n' and 'sha1' represent the algorithms employed by the model for the symbolic signing of the envelopes. 'sigvalA', 'sigvalSTS' and 'sigvalI' represent the signature values for Service A, the Security Token Service, and the intruder service, respectively. 'X509v3' represents the certificate version used in the signature. The XML Encryption constants used in the model are defined similarly. 'tripleDES' is the encryption algorithm used. For details about working with XML Signature and its algorithms

please refer to [TCM02]. The WS-Security Security Token Protocol contains the constant values of the public key and private key pairs used by Service A, the Security Token Service, and the intruder.

The mtype constants representing the WS-Trust specification parameters include 'partialSC' and 'SCT', which represent the partial context and full security context agreed on by the participants. 'claimsA' and 'claimsS' are claims made by Service A and the Security Token Service. 'RST' and 'RSTR' represent the Request Security Token and Response to Security Token Request in the envelopes. 'created' and 'expires' represent when the token was created and when the token will become invalid. These values represent the freshness values for the token, but not the message. The freshness of the message is represented by TimeStamp values. 'client_entropy' and 'server_entropy' are provided by the client and server, respectively, for calculating keys. The client and server entropies are used to establish a context key between services and are used as a session key. 'SCT_ID' represents the unique identity of the security context token.

Channels

Service A, the Security Token Service, and the intruder service use three channels to communicate with each other. The channel 'rstChan' is used by Service A to send request security token messages to the Security Token Service. The channel sends/receives messages of type {UsernameToken, recvr, Signature, RST, RequestSecurityToken, TimeStamp, EncryptedData, publicKey}. UsernameToken, Signature, RequestSecurityToken, TimeStamp and EncryptedData represent the 'typedef' structure explained in the global

variables section below. 'recvr' represents the username of the receiving entity, which is the Security Token Service. The 'publicKey' is the public key used for symbolic encryption of the message. The message is symbolically encrypted with the Security Token Service's public key. 'RST' is used as a message identifier for the request for security token.

The second channel 'rstrChan' is used by the Security Token Service to send the response to the security token request. The channel sends/receives messages of the form {UsernameToken, UsernameToken, RSTR, Signature, SC, RequestSecurityTokenResponse, TimeStamp, EncryptedData, publicKey}. 'UsernameToken' represents the username of the requesting service, the second 'UsernameToken' represents the Security Token Service. 'Signature', 'RequestSecurityTokenResponse', 'TimeStamp' and 'EncryptedData' represent the parameters for WS-Security and WS-Trust. 'RSTR' and 'SC' are the message identifier for the response to security token request and the security context. 'pubKey' represents the public key used for encryption.

The third channel, 'ackChan', is used by the requesting service to send the unique security token identifier back to the Security Token Service. This is done to allow the Security Token Service to verify the identity of the security context token it sent to the requesting service. The definitions of the channels are listed as follows.

```
chan rstChan = [0] of {UsernameToken, mtype, Signature, mtype,
RequestSecurityToken, TimeStamp, EncryptedData, mtype};

chan rstrChan = [0] of {UsernameToken, UsernameToken, mtype, Signature, mtype,
RequestSecurityTokenResponse, TimeStamp, EncryptedData, mtype};
```

chan ackChan = [0] of {mtype};

Global Variables

Global variables are used to represent the properties of the system. The properties of the Security Token Protocol, are represented as Linear Temporal Logic formulas. The protocol aims to issue a security token. 'typedef' complex structures used in our model are also defined.

There are six global variables used for checking the properties of the system: 'SenderBindA_S', 'SenderChallengeA_S', 'RecvrBindA_S', 'RecvChallengeA_S', 'partial_SC' and 'security_context'. When Service A initiates a message exchange with the Security Token Service, 'SenderChallengeA_STS' is set to true when Service A knows it is talking to the Security Token Service. Similarly, on the Security Token Service side, 'RecvrChallengeA_STS' is set to true when the Security Token Service knows it is talking to Service A. The initial messages confirm for both Service A and the Security Token Service that they are in a run with each other. Only when Service A is ready for further exchange of messages with the Security Token Service does it set 'SenderBindA_S' to true. Similarly, after confirming the identity of the sender the Security Token Service commits to it, and sets 'RecvrBindA_S' to true. 'partial_SC' is the global variable, updated once by the Security Token Service, indicating whether an agreement has been reached on claims made by the requesting service. 'partial_SC' represents a combination of elements, such as 'AppliesTo' (the service the security token is valid for, in our case the Security Token Service), 'TokenType' (the type of token being requested, the security context token), 'RequestType' (the type of request, issuing a security token), 'EntropicMode' (entropic mode used for calculating keys – we

use partial entropy where both the requestor service and Security Token Service provide their entropies), and 'client_entropy' (the entropy provided by the client for computing keys). The global variable 'Security_Context' is true on the requestor side when it agrees on the security context information provided by the Security Token Service. It is set true when the 'partial_SC' is true and the Security Token Service has provided additional information, i.e., 'server_entropy' (the entropic value provided by the server to calculate the security context key), 'expires' (how long the token is valid for), 'stsnonce' (nonces are freshly generated random values), the Security Token Service's identity, the requestor's identity, and a unique identifier for the security context token. The values of these variables are updated using macros in the model. The updated values are then used for verifying the correctness properties of the system.

We now model the above global variables with Linear Temporal Logic formula.

[] (([] !senderbindAS) || (!senderbindAS U recvrchallengeAS))

[] (([] !recvrbindAS) || (!recvrbindAS U senderchallengeAS))

[] (([] !securitycontext) || (!securitycontext U partialSC))

The above formulas represent the property requirements the system is to satisfy. The formula reads as (i) 'senderbindAS' is always false or 'senderbindAS' is false until 'recvrchallengeAS' becomes true; and (ii) 'recvrbindAS' is always false or 'recvrbindAS' remains false until 'senderchallengeAS' becomes true. Initially, the 'securitycontext' is always false, or 'securitycontext' becomes true when there is an agreement on a partial context 'partialSC' between a

requestor and a Security Token Service.

Tables 5.5 and 5.6 represent typedef structures used in the Security Token Protocol. Table 5.5 contains complex structures for XML Signature, XML Encryption, TimeStamp and UsernameToken. Table 5.6 represents the complex structures for requesting a security token and for the response to a security token request.

Principal Processes

The Security Token Protocol consists of three principals, a token requesting service (Service A), a Security Token Service (service STS), and an intruder service. The Promela model of the Security Token Service consists of four processes in total: the 'init' process, 'sender' process, 'STS' process and 'intruder' process.

Init process: The 'init' process contains the instantiation statements for the 'sender', 'STS' and 'intruder' processes. The model alternates between an 'intruder' process and a 'sender' process at a given time. The STS process is always running and awaiting token issuance requests from the requestor. The intruder process has three parameters passed to it: the identity of the intruder, the service it wants to communicate with (in this case the Security Token Service), and its nonce, nonceI. Similarly, the 'sender' process passes its username A, the service it wants to talk to, 'STS', and its freshly generated nonce, nonceA. The 'STS' service is passed its username STS and its freshly generated nonce, nonceS.

Table 5.5: Security Token Protocol Global Variables

TimeStamp	typedef TimeStamp{ mtype Created; mtype Expires; }; TimeStamp ts[6];
XML Signature	typedef Signature { mtype CanonicalizationMethod ; mtype SignatureMethod ; mtype Reference ; mtype SignatureValue ; mtype KeyInfo; }; Signature sig[6];
XML Encryption - Encrypted Data	typedef EncryptedData{ mtype EncryptionMethod ; mtype CipherData ; mtype CipherValue ; mtype ReferenceList ; mtype DataReference ; mtype KeyInfo; }; EncryptedData edata[6];
XML Encryption - Encrypted Key	typedef EncryptedKey { mtype EncryptionMethod ; mtype CipherData ; mtype CipherValue ; mtype ReferenceList ; mtype DataReference ; mtype KeyInfo; }; EncryptedKey ek[6];
UsernameToken	typedef UsernameToken { mtype Username; mtype Nonce; }; UsernameToken ust[7];

Table 5.6: Security Token Protocol Global Variables

RequestSecurityToken	typedef RequestSecurityToken{ mtype TokenType ; mtype RequestType ; mtype AppliesTo; mtype Entropy; mtype EntropicMode; mtype rst_created; mtype rst_expires; }; RequestSecurityToken rst[2];
RequestSecurityToken	typedef typedef RequestSecurityTokenResponse{ mtype TokenType ; mtype RequestType ; mtype AppliesTo; mtype Entropy; mtype EntropicMode; mtype ComputedKey; mtype rstr_created; mtype rstr_expires; mtype sct_id; }; RequestSecurityTokenResponse rs[2];

```
init {
    if
      ::run Intruder(I, STS, nonceI);
      ::run Sender(A, STS, nonceA);
    fi;
    run sts(STS, nonceS);
}
```

'Sender' Process: The 'sender' process is a security token requestor service. The 'sender' process sends a request to the Security Token Service, to issue a security token. The request for a security token comprises of three main steps as described in the automaton model in the Chapter 4. First, the 'sender' creates a request for a security token and sends it to the Security Token Service. Second, it waits for the response for issuing the security token and processes the response when it arrives. Third, if it accepts the security context token, it sends a security context id back to the Security Token Service; otherwise it sends a 'MSG_REJ' to the Security Token Service.

The 'sender' sends a request for issuing a security token to the 'STS' process. It populates the 'typedef' structures specified in Tables 5.5 and 5.6. It sends its UsernameToken containing its username, and its nonce to the receiver process 'STS', as well as the XML Signature information used to sign the request. The request contains all the information to obtain a partial security context agreement between Service A and the Security Token Service. The message required also contains the TimeStamp information, as well as information used to encrypt the data. The message is encrypted using the public key of the Security Token Service.

The 'sender' process waits for the response from 'STS' on 'rstrChan'. It receives its own UsernameToken and the 'STS' UsernameToken, along with the response to the security token request. The XML Signature and XML Encryption information are used to sign and encrypt the message by the requesting service. The 'sender' processes the incoming message. It decrypts the message using its private key, and checks the TimeStamp token as well as the Encryption and Signature information. It validates that it is an RSTR packet, and then commits to the session between 'STS' and itself, if it validates that it is in a run with 'STS', and finally agrees on a security context agreement. The security context agreement is reached if both the parties agree the on following information: AppliesTo, TokenType, RequestType, EntropicMode, Entropy of both server and client, ComputedKey, Expiry of the token, and the freshness of the nonces.

```
proctype Sender(mtype me; mtype recvr; mtype my_nonce)
{
    atomic {
        (recvr == STS)→ pubKey = pubKeySTS
        SenderChallenge(me, recvr);
        rstChan!ust[0],recvr,sig[0],RST,rst[0],ts[0],edata[0],pubKey ;
    }
    atomic {
        rstrChan?ust[4],ust[3],rstr,sig[3],SC, rs[1],ts[3],edata[3],pubKey;
        Decryption(pubKey, privateKey);
        VerifyFreshness(ts[3].Created,ts[3].Expires);
        VerifySignature(sig[3].CanoncalizationMethod, sig[3].SignatureMethod,
            sig[3].Reference, sig[3].SignatureValue, sig[3].KeyInfo);
        AuthenticateResponse(rstr);
        SenderBind(ust[4].Username, ust[3].Username);
        SCAgreement(rs[1].AppliesTo, rs[1].TokenType, rs[1].RequestType,
```

```
            rs[1].EntropicMode, rst[0].Entropy, rs[1].Entropy,rs[1].ComputedKey,
            rs[1].rstr_expires, ust[3].Nonce, ust[3].Username, ust[4].Username, rs[1].sct_id);
    }
    bit FLAG = 1;
    mtype msg;
    if
        :: (Security_Context == FLAG) → msg = SC
        :: (Security_Context != FLAG) → msg = MSG_REJ
    fi;
    ackChan!msg;
}
```

'STS' Process: The 'STS' process is a Security Token Service issuing process. The 'STS' process is responsible for listening for security token requests and relaying a security token response back to the requestor. The Security Token Service listens for requests for a security token on 'rstChan'. When the Security Token Service receives a request it decrypts the message, verifies the freshness of the message, and the signature information of the sender, updates it global variables, and agrees on a partial security context. It creates a response that is to be sent back to the requester. It assigns values to the 'typedef' structures listed in Tables 5.5 and 5.6. It creates a request security token response, containing the security token, and sends it to the sender service on channel 'rstrChan'. The Security Token Service then waits for an acknowledgement containing the security context ID on 'ackChan'. The 'STS' process is listed as follows.

```
proctype sts(mtype me; mtype my_nonce)
{
    atomic {
        rstChan?ust[1],eval(me),sig[1],req,rst[1],ts[1] edata[1],pubKey;
```

```
        Decryption(pubKey, privateKey);
        VerifyFreshness(ts[1].Created,ts[1].Expires);
        VerifySignature(sig[1].CanoncalizationMethod, sig[1].SignatureMethod,
            sig[1].Reference, sig[1].SignatureValue, sig[1].KeyInfo);
        RecvrChallenge(ust[1].Username, me);
        AuthenticateRequest(req);
        RecvrBind(ust[1].Username, me);
        PartialSCAgreement(rst.AppliesTo, rst.TokenType, rst.RequestType,
            rst.EntropicMode, rst.Entropy);
        if
            :: (ust[1].Username == A)→ pubKey = pubKeyA
            :: (ust[1].Username == I) → pubKey = pubKeyI
        fi;
        rstrChan!ust[1],ust[2],RSTR,sig[2],SC,rs[0],ts[0],edata[2],pubKey;
    }
    mtype x;
    ackChan? x
}
```

'Intruder' Process: The intruder behaves in the following ways. It can act as a new user and gather information represented by 'I →STS'. It can block and intercept a message from sender A, and pass it to the Security Token Service 'I(A) → STS'. In this scenario no signature or information is leaked to the intruder.

The intruder is a constantly running process. When the intruder process is initiated, it populates the 'typedef' structures in Tables 5.5 and 5.6. It assigns values to all structures apart from 'RequestSecurityTokenResponse'. In 'I →STS', the intruder acts as a legitimate user of the system and sends a request for a security context token to the Security Token Service on channel 'rstChan'. It then waits for a response from the Security Token Service

on channel 'rstrChan'. It processes the request and establishes a security context. It then sends the security context identity back to the sender on 'ackChan'. This security context can be later used in establishing a session with other unsuspecting services.

The intruder learns Username token information from runs with Service A and sends requests to the Security Token Service: 'I (A) → STS'. The intruder blocks messages sent from Service A and forwards them to the Security Token Service. It then waits for a response from the Security Token Service. If the message is authenticated at the server end, it receives a security context token, or if the attack is identified it receives a message reject response from the server. The intruder process is listed as follows.

```
proctype Intruder(mtype me; mtype recvr; mtype my_nonce){
    do
    :: atomic {
        atomic {
            pubKey = pubKeySTS;
            SenderChallenge(me, recvr);
            rstChan!ust[5],recvr, sig[4],RST,rst[0],ts[0],edata[4],pubKey ;
        }
        atomic {
            rstrChan?ust[5],ust[6],rstr,sig[5],SC,rs[1],ts[1],edata[5],pubKey;
            Decryption(pubKey, privateKey);
            VerifyFreshness(ts[1].Created,ts[1].Expires);
            VerifySignature(sig[5].CanonicalizationMethod, sig[5].SignatureMethod,
                sig[5].Reference, sig[5].SignatureValue, sig[5].KeyInfo);
            AuthenticateResponse(rstr);
            SenderBind(ust[5].Username, ust[6].Username);
            SCAgreement(rs[1].AppliesTo, rs[1].TokenType, rs[1].RequestType,
```

```
            rs[1].EntropicMode, rst[0].Entropy, rs[1].Entropy,rs[1].ComputedKey,
            rs[1].rstr_expires, ust[6].Nonce, ust[6].Username, ust[5].Username, rs[1].sct_id);
    }
    bit FLAG5 = 1; mtype msg5;
    if
        :: ((Security_Context_I == FLAG5) && (valid_DecryptI == FLAG5))→ msg5 = SC
        :: (Security_Context_I != FLAG5) → msg5 = MSG_REJ
    fi;
    ackChan!msg5;
}
:: atomic{
    atomic {
        pubKey = pubKeySTS;
        SenderChallenge(ust[0].Username, recvr);
        rstChan!ust[0], recvr,sig[4],RST,rst[0],ts[0],edata[4],pubKey;
    }
    atomic {
        rstrChan?ust[5],ust[6],rstr,sig[5],SC,rs[1],ts[1],edata[5],pubKey;
        Decryption(pubKey, privateKey);
        VerifyFreshness(ts[1].Created,ts[1].Expires);
        VerifySignature(sig[5].CanoncalizationMethod, sig[5].SignatureMethod,
            sig[5].Reference, sig[5].SignatureValue, sig[5].KeyInfo);
        AuthenticateResponse(rstr);
        SenderBind(ust[5].Username, ust[6].Username);
        bit FLAG4 = 1; mtype msg4;
        if
        ::((valid_dsSigA == FLAG4)&&(valid_DecryptA == FLAG4))
            →SCAgreement(rs[1].AppliesTo, rs[1].TokenType, rs[1].RequestType, rs[1].EntropicMode,
            rst[0].Entropy, rs[1].Entropy,rs[1].ComputedKey, rs[1].rstr_expires,
            ust[3].Nonce, ust[3].Username, ust[4].Username, rs[1].sct_id);
        ::else → ackChan!MSG_REJ
        fi;
        if
```

```
            :: (Security_Context == FLAG4) → ackChan! SC;
            fi;
        }
    }
    od
}
```

Macros are used in the Security Token Protocol to update the values of the global variables used in the Linear Temporal Logic formulas. The 'SenderChallenge(X,Y)', 'SenderBind(X,Y)', 'RecvrChallenge(X,Y)' and 'RecvrBind(X,Y)' macros work in the same way as the ones described for the Simple Message Exchange Protocol. Two new macros are introduced here: 'PartialSCAgreement(a,b,c,x,y)' and 'SCAgreement(at,tt,rt,em,ce,se,ck,ex,stsn,sts,sdr,scid)'. These macros update the values of global variables 'partial_SC' and 'Security_Context'. The macro 'PartialSCAgreement(a,b,c,x,y)' is used to reach an agreement on a partial security context, based on the elements AppliesTo, TokenType, RequestType, EntropicMode and Entropy. If all the elements are fulfilled, it sets 'partial_SC' to true. Similarly,the macro 'SCAgreement(at,tt,rt,em,ce,se,ck,ex,stsn,sts,sdr,scid)' is used for an agreement on a full security context. The full context is agreed if AppliesTo, TokenType, RequestType, EntropicMode, ClientEntropy, ServerEntropy, ComputedKey, Expires, UsernameToken for STS and A, and SC id are valid, in which case 'Security_context' is set to true. Other macros are also used in the Security Token Protocol.

```
#define PartialSCAgreement(a,b,c,x,y) if
    :: ((a== STS)&&(b== SCT)&&(c== ISSUE)&&(x== partialEntropy)&&
          (y== client_entropy)) → partial_SC=1
    :: else skip
```

```
    fi

#define SCAgreement(at,tt,rt,em,ce,se,ck,ex,stsn,sts,sdr,scid) if
    :: ((at == STS)&&(tt == SCT)&&(rt == ISSUE)&&(em == partialEntropy)&&
        (ce == client_entropy)&&(se == server_entropy)&&(ck == sha1)&&
        (ex == expires)&&(stsn == nonceS)&&(sts == STS)
        &&(sdr == A)&&(scid == SCT_ID)) -> Security_Context=1
    ::else skip;
    fi
```

5.4 Concluding Remarks

This chapter has dealt with the modelling into Promela of the protocols defined earlier. The behaviour of the principals involved in the protocol run, i.e., Service A, Service B, and the Security Token Service, have been identified. An intruder model based on the Dolev-Yao model has been presented in Promela. In the next chapter simulations are performed to verify the general behaviour of the protocols by inputting these models to XSpin, a graphical interface to the *Spin* model checker.

CHAPTER 6

Simulation and Verification Results

To understand the behaviour of the model of a system, and the properties the model captures, simulations are performed on the model. Simulation allows the detection of any deviation from the expected behaviour of the system. Prototypes have been developed for the Simple Message Exchange Protocol and the Security Token Protocol, and *Spin* [Hol03, BK08] has been used for rapid prototyping.

XSpin, the graphical interface to *Spin*, is used for simulation and verification purposes. It runs *Spin* commands in the background. XSpin is used for model checking the Simple Message Exchange Protocol and the Security Token Protocol. XSpin provides three simulation modes: random, guided and interactive simulation. Random simulation is carried out using a predefined seed value. Different runs can be obtained by changing the seed value. The

guided simulation requires the presence of a 'pan.trail' file, produced during the verification run. In this thesis interactive simulation is used to resolve manually non-deterministic choices in the model. A choice is offered when there are different directions the execution can proceed in.

There are four basic types of output that can be requested: *(i)* Message Sequence Charts, *(ii)* Time Sequence, *(iii)* Data Value, and *(iv)* Execution Bar. Briefly, a message sequence chart displays send and receive actions, connecting matching pairs with arrows. The Time Sequence option shows text output of the simulation run. The Data Value option shows the most recent values assigned to the variables in the model. The Execution Bar option gives a dynamically updated bar-chart of the number of statement executions in each running process.

6.1 Chapter Objectives

This chapter deals with two things: simulation and verification of the protocol models. Simulations are performed for each scenario discussed in Section 3.6 for the Simple Message Exchange Protocol and the Security Token Protocol. Simulation allows us to analyse the behaviour of the model and correct any mistakes for further verification. Message Sequence Charts and Data Values were selected as the output modes for simulation runs. Using message sequence charts and data values enables the tracking of the exchange of messages between protocol participants.

Verification of the protocol models is performed to show that the system satisfies certain properties. The properties of secrecy and authentication are

verified for the Simple Message Exchange Protocol. In the case of the Security Token Protocol the properties of secure agreement of a security context while maintaining secrecy and authentication goals are verified. These properties are represented in Linear Temporal Logic formulas.

6.2 Simulation Results

6.2.1 Simple Message Exchange Protocol

The Simple Message Exchange Protocol (SMEP) exchanges three messages between two services, a requestor service and a responder service. At the end of a valid run, the two services have securely authenticated to each other. The challenge is to achieve the authenticated state under the presence of an active intruder. The possible manipulated protocol runs are modelled for SMEP as described in Chapter 3. The intruder service can interact with other services in the following possible ways. First, the intruder service I, can initiate a conversation with Service B in order to learn information about Service B: 'I→B'. Secondly, the intruder service can act as a legitimate user and be involved in a protocol run with Service A: A →I. Thirdly, the intruder service after learning information about Service B can act as service B, and be involved in a protocol run with Service A: A→I(B). Fourthly, the intruder service can act as Service A and initiate a run with service B: I(A)→B. Simulation is performed for a behavioural analysis of the protocol, and we suggest changes accordingly. Results are presented in the following sections.

A → B

A correct Simple Message Exchange Protocol run is successfully achieved when Service A and Service B are able to authenticate to each other. Figure 6.1 represents a snapshot of a message sequence chart for the protocol run. The processes representing Service A, Service B and the intruder service are represented as boxes, and the numbers represent the execution steps in the simulation run. The message sequence chart shows send and receive actions, connecting matching pairs with arrows. Sender A sends a message {A, nonceA, B, REQ, ts[0], sig[0], edata[0], pubKeyB} to Service B in execution step 40. 'ts[0]', 'sig[0]' and 'edata[0]' represent the timestamp, signature and encryption information for Service A. Sender A updates its global variable 'senderchallengeAB' to 1. The message is processed at the receiver, Service B. The receiver updates the global variable 'recvchallengeAB' to 1 before sending a response back to Service A. The message sent is {B, nonceB, A, nonceA, ACK, ts[2], sig[2], edata[2], pubKeyA}in step 53. 'ts[2]', 'sig[2]' and 'edata[2]' represent the timestamp, signature and encryption information for Service B. Service A validates the message and the identity of Service B, and then updates 'senderbindAB' to 1. It then creates an acknowledgement message for Service B. The message contains {A, nonceA, B, nonceB, ACCEPT, TimeStamp, Signaturelnfo, EncryptedData, pubKey(B)}, and is sent in execution step 67. The receiver processes the message and validates the identity of the sender and authenticates it by updating 'recvrbindAB' to 1. The successful run in the end shows the values of all the global variables 'senderchallengeAB', 'senderbindAB', 'recvrchallengeAB' and 'recvrbindAB' to be 1. The global variables are updated using macros defined in the Promela model as explained in Chapter 5. Service A binds to a session with Service B only when it is engaged in a session with B. Similarly, B only binds to A, when

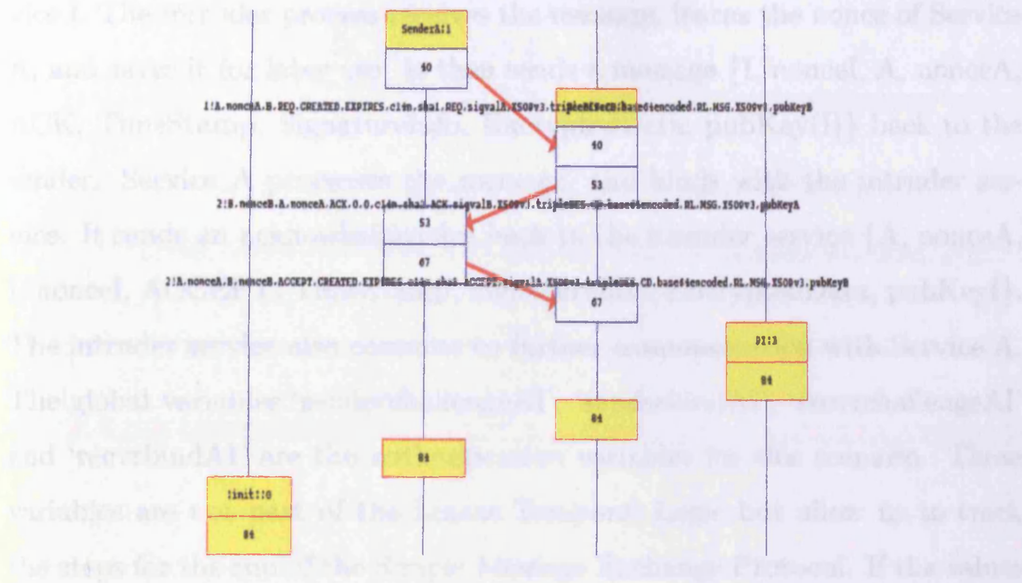


Figure 6.1: Message sequence chart for A $\rightarrow$ B

it knows that it is engaged in a run with A. Global variables called 'secrecy' variables are defined. Their values are updated when the XML Encryption and XML Signature values are satisfied. 'valid_DecryptA' is updated to 1 when all the information necessary to decrypt the message received from B is present in the message. Similarly, 'valid_DecryptB' is updated to 1 when the message can be decrypted by the corresponding symbolic decryption values of B. 'valid_TimeStamp' is a timestamp variable, whose value is updated when the TimeStamp attributes are present in the message.

A $\rightarrow$ I

Intruder I acts a legitimate user of the protocol. Service A initiates an SMEP run with service I, thinking it is legitimate user. It sends a message {A, nonceA, I, REQ, TimeStamp, SignatureInfo, EncryptedData, pubKeyI} to ser-

vice I. The intruder process receives the message, learns the nonce of Service A, and saves it for later use. It then sends a message {I, nonceI, A, nonceA, ACK, TimeStamp, SignatureInfo, EncryptedData, pubKey(I)} back to the sender. Service A processes the message, and binds with the intruder service. It sends an acknowledgement back to the intruder service {A, nonceA, I, nonceI, ACCEPT, TimeStamp, SignatureInfo, EncryptedData, pubKeyI}. The intruder service also commits to further communication with Service A. The global variables 'senderchallengeAI', 'senderbindAI', 'recvrchallengeAI' and 'recvrbindAI' are the authentication variables for this scenario. These variables are not part of the Linear Temporal Logic but allow us to track the steps for the run of the Simple Message Exchange Protocol. If the values of the above global variables are all 1, the protocol run has been successful and the intruder service has had a successful run with Service A as a legitimate user. The other global variables monitored are 'valid_DecryptA', 'valid_DecryptI' and 'TimeStamp'. Their values are all updated to 1 after a successful run. When the intruder process learns the nonce of Service A, it updates its knowledge base and sets the global variable 'learn_kNa' to 1.

Figure 6.2 gives a snapshot message sequence chart for the run. 'init', 'SenderA', 'ReceiverB' and 'PI' are the four processes represented in the figure. In execution step 45 a message {A, nonceA, I, REQ, ts[0], sig[0], edata[0], pubKeyI} is sent from SenderA to PI. The intruder processes the message, and in execution step 56 it sends the message {I, nonceI , A, nonceA, ACK, ts[1], sig[1], edata[1], pubKeyA} to SenderA. SenderA processes the request and in execution step 75 sends the message {A,nonceA, I, nonceI, ACCEPT, ts[0], sig[0], edata[0], pubKeyI} to the intruder process. The structures 'ts[index]', 'sig[index]' and 'edata[index]' represent the timestamp,

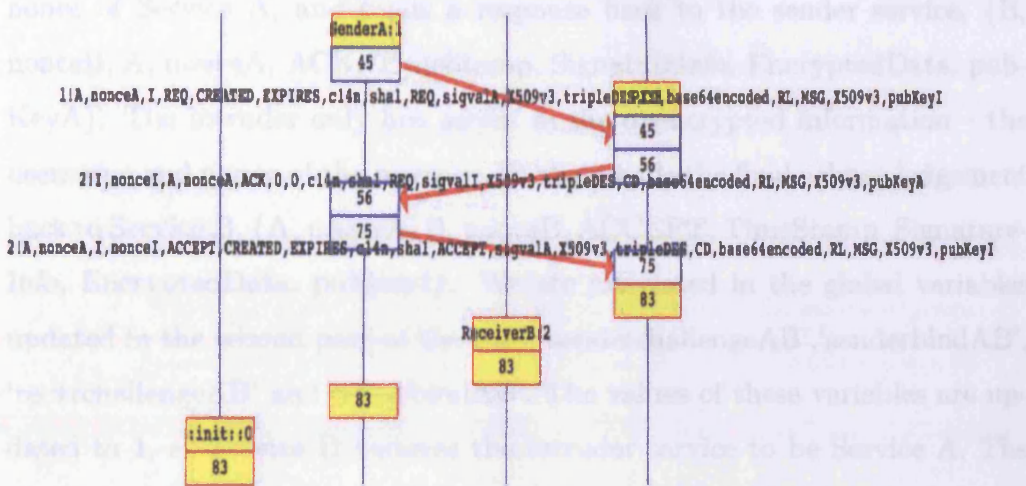


Figure 6.2: Message Sequence Chart for A→I

signature and encryption values for the different services.

A → I followed by I(A) → B

The first part of this scenario is similar to the one above, and Service A interacts with the intruder service, assuming it to be a legitimate user of the system. During this run, the intruder process learns the nonce of Service A and immediately initiates an attack on Service B, pretending to be Service A. The intruder can only execute this sequence of events if its knowledge base contains the nonce of Service A, i.e., its 'learn_kNa' is set to 1. The intruder creates the following message to send to Service B on the channel: {A, nonceA, B, REQ, TimeStamp, SignatureInfo, EncryptedData, pubKeyB}. The receiver processes the request, and it sees the fresh timestamp and a unique

nonce of Service A, and sends a response back to the sender service, {B, nonceB, A, nonceA, ACK, TimeStamp, Signaturelnfo, EncryptedData, pubKeyA}. The intruder only has access to the unencrypted information – the username and nonce of the message. It then sends the final acknowledgement back to Service B, {A, nonceA, B, nonceB, ACCEPT, TimeStamp, SignatureInfo, EncryptedData, pubKeyl}. We are interested in the global variables updated in the second part of the run: 'senderchallengeAB','senderbindAB', 'recvrchallengeAB' and 'recvrbindAB'. The values of these variables are updated to 1, as Service B believes the intruder service to be Service A. The simulation run shows that if the intruder learns the nonce of Service A, it can make Service B believe that it is authenticating to Service A. The Decrypt variables are updated, and the intruder service acting as I(A) sends the message it has intercepted from A to Service B. Service B decrypts the message using its private key and updates 'valid_DecryptB' to 1. Service B creates a response for I(A), and encrypts the message with A's public Key. The intruder, on receiving the message from B is unable to decrypt the message as it has no knowledge of A's private key. It can save the message or discard it.

Figure 6.3 gives a snapshot of the scenario. Execution steps 45, 56 and 75 are similar to the ones explained in the previous section. The intruder sends a message {A, nonceA, B ,REQ, ts[0], sig[0], edata[1], pubKeyB } to ReceiverB acting as Service A in step 85. ReceiverB processes the request and sends the message {B, nonceB, A, nonceA, ACK, ts[2], sig[2], edata[2], pubKeyA} back to the intruder (which is pretending to be service A) in execution step 100. The intruder sends a response message {A, nonceA, B, nonceB, ACCEPT, TimeStamp, Signaturelnfo, EncryptedData, pubKey(B)} in step 114.

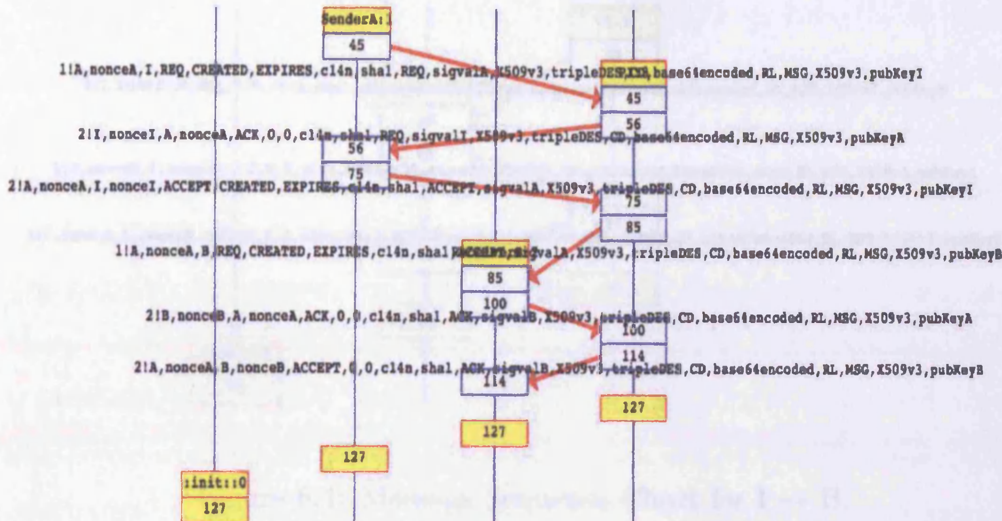


Figure 6.3: Message Sequence Chart for A → I followed by I(A) → B.

I → B

The intruder service masquerades as a legitimate service, and initiates a Simple Message Exchange Protocol run with Service B. The intruder sends the message {I, nonceI, B, REQ, TimeStamp, SignatureInfo, EncryptedData, pubKeyB} to Service B. Service B sends a response to the intruder thinking it as a valid service. It sends {B, nonceB, I, nonceI, ACK, TimeStamp, SignatureInfo, EncryptedData, pubKeyI} back to the intruder service, and the intruder returns the final accept message back to Service B, {I, nonceI, B, nonceB, ACCEPT, TimeStamp, SignatureInfo, EncryptedData, pubKeyB}, resulting in Service B committing to further message exchange with the intruder service. Figure 6.4 shows a message sequence chart for the run. The intruder service learns the nonce of Service B, and updates 'learn_kNb' to 1.

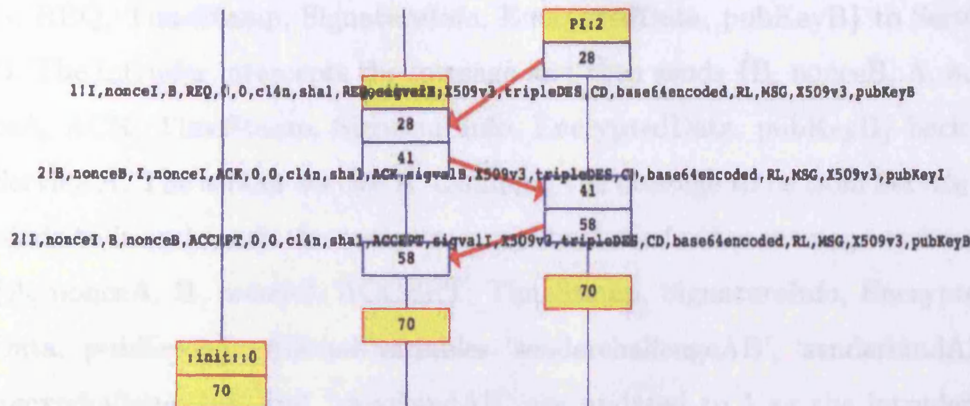


Figure 6.4: Message Sequence Chart for I → B.

Figure 6.4 shows a snapshot of the message exchange between the intruder and Service B. In step 28, 'PI' sends a message {I, nonceI, B, REQ, ts[1], sig[1], edata[1], pubKeyB} to 'ReceiverB'. 'ReceiverB' processes the message and sends a response message {B, nonceB, A, nonceI, ACK, ts[2], sig[2], edata[2], pubKeyI} back to 'PI' assuming it to be a non-threatening service. 'PI' ends the protocol run in execution step 58 by sending a final accept message {I, nonceI, B, nonceB, ACCEPT, ts[1], sig[1], edata[1], pubKeyB} back to 'ReceiverB'.

I → B followed by A → I(B)

This scenario is similar to 'A→ I followed by I(A) → B', but in this case, the intruder process learns information about Service B by initiating a Simple Message Exchange Protocol run, and subsequently imitates Service B. The intruder intercepts and processes any messages for Service B from Service A, and masquerades as Service B itself. Sender A sends a message {A, nonceA,

B, REQ, TimeStamp, SignatureInfo, EncryptedData, pubKeyB} to Service B. The intruder intercepts the message and then sends {B, nonceB, A, nonceA, ACK, TimeStamp, SignatureInfo, EncryptedData, pubKeyB} back to Service A. The sender service A, assuming the message to be from Service B, binds to it and sends the accept message back for further message exchange: {A, nonceA, B, nonceB, ACCEPT, TimeStamp, SignatureInfo, EncryptedData, pubKeyA}. Global variables 'senderchallengeAB', 'senderbindAB', 'recvrchallengeAB' and 'recvrbindAB' are updated to 1 as the intruder is able to successfully attack the protocol. During run 'A → I(B)', the decryption variable DecryptB is not updated, as the intruder does not possess the private key of B to decrypt the messages. The valid_DecryptB value represents the last updated value during the run 'I → B'.

Figure 6.5 shows a snapshot of the message sequence chart for the run. Steps 45, 58 and 75 are same as the ones described in the last scenario. Steps 87, 105 and 119 show a run of the protocol under attack.

The simulation results are summarised in Table 6.1 for the Simple Message Exchange Protocol. The table lists the values of the global variables 'senderBindAB', ' recvrBindAB', 'learn_kNa' and 'learn_kNb'. The value 0 for 'senderBindAB' and 'recvrBindAB' represents failure in authentication between Service A and Service B. The values of 'learn_kNa' and 'learn_kNb' when set to 0 shows that the intruder has not learned the nonce of Service A and Service B, respectively. 'Encryption' is set to 0 when encryption has not been broken. It can be seen that for each run in which the principals are involved with the intruder, the intruder learns the nonces. The intruder then uses these nonces and the identity of the principal to bind to another protocol

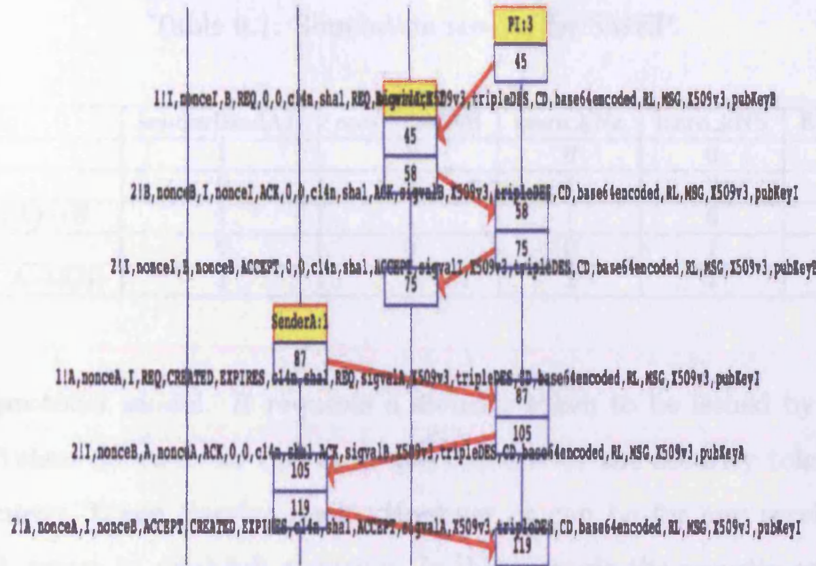


Figure 6.5: Message Sequence Chart for I → B followed by A → I(B).

participant. It can also be seen that the encryption remain unbroken, as the intruder does not have the capability to break the encryption. It can be seen that encrypting the UsernameToken can prevent the intruder from learning nonce information. However, in some cases it may lead to unforeseen complications, and so it may not be desirable to encrypt such information.

6.2.2 Security Token Protocol

This section presents the results obtained during simulation runs for the Security Token Protocol (STP). The participants for this protocol are process A, process I, and a Security Token Service, STS. The STS process cannot initiate any message exchange. Its sole purpose is to issue security tokens based on requests made by processes A and I. Process A is a legitimate user

Table 6.1: Simulation results for SMEP.

Scenario	senderBindAB	recvrBindAB	learn_kNa	learn_kNb	Encryption
A→B	1	1	0	0	0
A→I	0	0	1	0	0
A→I, I(A)→B	1	1	1	0	0
I→B	0	0	0	1	0
I → B, A→I(B)	1	1	1	1	0

of the protocol model. It requests a security token to be issued by the Security Token Service. In this case, the request for the security token is for the Security Token Service itself. However, it can be for any service with which A wants to establish a session. In this scenario the security context is represented in the form of a Request Security Token, as this seems logical in wanting to build towards WS-SecureConversation. There are three possible simulation scenarios for this protocol model, as discussed below.

A → STS

Figure 6.6 represents a snapshot of a message sequence chart for a valid run of the Security Token Protocol. The requested token is a security context token for establishing a session between Service A and the Security Token Service. The sender process sends a message of the form {UsernameToken, STS, Signature, RST, RequestSecurityToken, TimeStamp, EncryptedData, publicKeySTS} to the Security Token Service in step 43. The username token contains the username and the fresh nonce for Service A. The message contains all the information needed to establish a partial security context on the Security Token Service end. The Security Token Service returns a message containing the security token requested by the sender process in

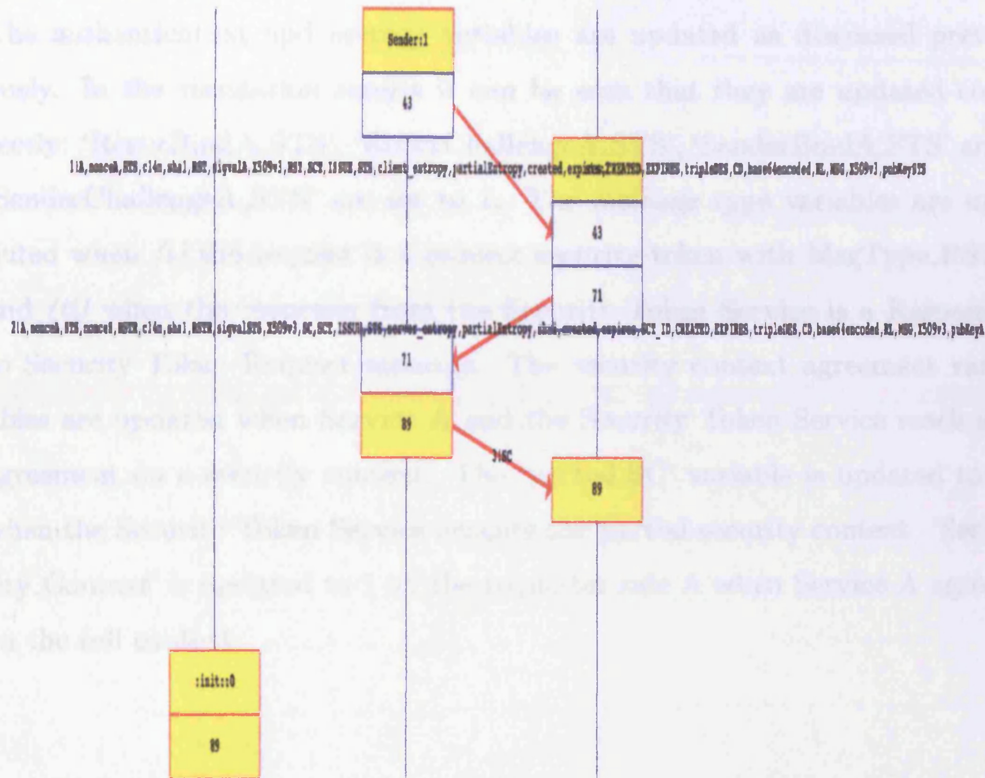


Figure 6.6: Message Sequence Chart for A → STS.

step 71. The message contains { UsernameToken(A, nonceA), UsernameToken(STS, nonceS), RSTR, Signature, SC, RequestSecurityTokenResponse, TimeStamp, EncryptedData, publicKeyA}. The sender service agrees on a full security context and sends the unique security context ID, {SC}, back to the Security Token Service in step 89.

There are four types of global variables of relevance in this scenario. The *(i)* authentication and *(ii)* secrecy variables, as discussed before, and

the *(iii)* message type and *(iv)* security context agreement global variables. The authentication and secrecy variables are updated as discussed previously. In the simulation results it can be seen that they are updated correctly: 'RecvrBindA_STS', 'RecvrChallengeA_STS', 'SenderBindA_STS' and 'SenderChallengeA_STS' are set to 1. The message type variables are updated when *(i)* the request is a request security token with MsgType_RST, and *(ii)* when the response from the Security Token Service is a Response to Security Token Request message. The security context agreement variables are updated when Service A and the Security Token Service reach an agreement on a security context. The 'partial_SC' variable is updated to 1 when the Security Token Service accepts the partial security context. 'Security_Context' is updated to 1 on the requester side A when Service A agrees on the full context.

I → STS

Figure 6.7 shows a snapshot of a message sequence chart for an intruder/impostor requesting a security token. The Security Token Service treats the intruder as a legitimate process and issues it a security token. The intruder process sends a request for a security token, {UsernameToken(I, nonceI), STS, Signature, RST, RequestSecurityToken, TimeStamp, EncryptedData, publicKeySTS}, in step 45. The Security Token Service processes the request and assumes the intruder to be a legitimate user and issues it a security context token. The Security Token Service sends the message {UsernameToken(I,nonceI), UsernameToken(STS, nonce), RSTR, Signature, SC, RequestSecurityTokenResponse, TimeStamp, EncryptedData, publicKeyI} to the intruder service in step 73. The intruder service returns the

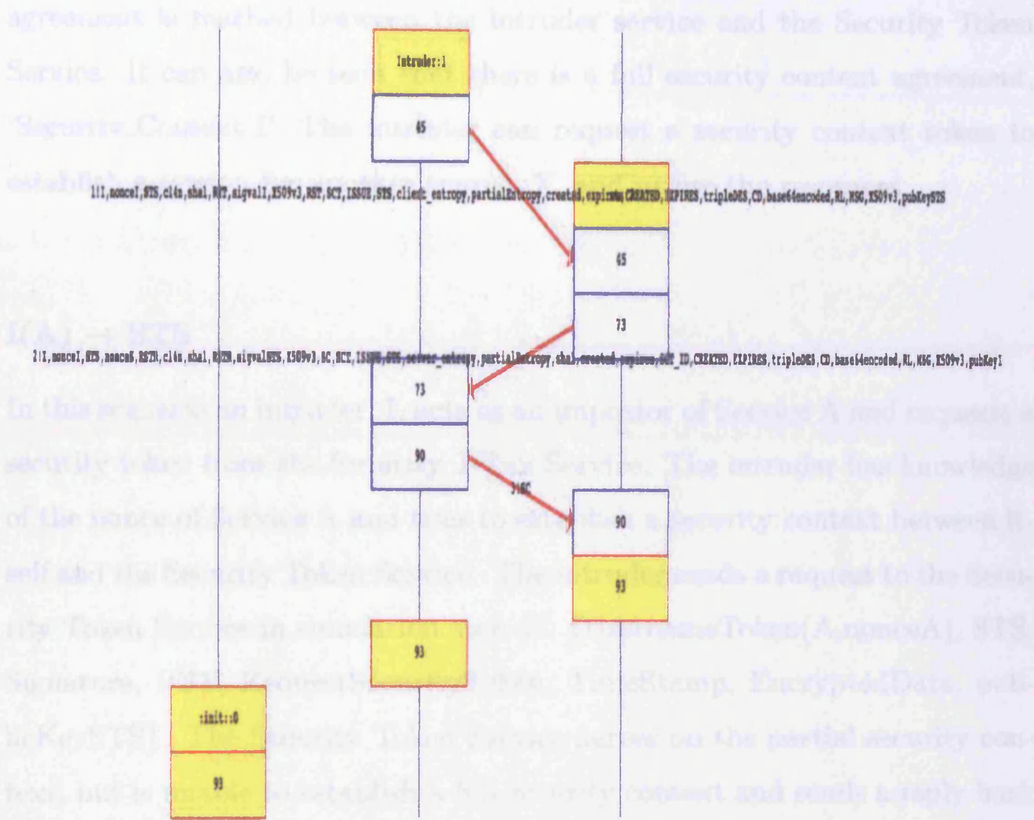


Figure 6.7: Message Sequence Chart for I → STS

identity of the security context received, {SC}, to the Security Token Service in step 90. The simulation results show a correct run between I and STS.

It can be seen from the global variables that the model runs correctly. The authentication, secrecy and message type variables for tracking the message exchange between the intruder and the Security Token Service are updated to 1 (‘SenderChallengeI_STS’, ‘SenderBindI_STS’, ‘RecvrBindI_STS’ and ‘RecvrChallengeI_STS’). It can be seen from ‘partial_SC’ that partial

agreement is reached between the intruder service and the Security Token Service. It can also be seen that there is a full security context agreement, 'Security_Context_I'. The intruder can request a security context token to establish a session for another source, X, and utilise the resources.

I(A) → STS

In this scenario an intruder, I, acts as an impostor of Service A and requests a security token from the Security Token Service. The intruder has knowledge of the nonce of Service A and tries to establish a security context between itself and the Security Token Service. The intruder sends a request to the Security Token Service in simulation step 45, {UsernameToken(A,nonceA), STS, Signature, RST, RequestSecurityToken, TimeStamp, EncryptedData, publicKeySTS}. The Security Token Service agrees on the partial security context, but is unable to establish a full security context and sends a reply back to the sender, {UsernameToken(A, nonceA), UsernameToken(STS, nonceS), RSTR, Signature, SC, RequestSecurityTokenResponse, TimeStamp, EncryptedData, publicKeyA}, in step 73. The intruder masquerading as Service A fails to establish a security context with the Security Token Service, and sends {MSG_REJ} in step 87. This is indicated by the value of the global variable 'Security_Context' which is not updated to 1 as the encryption remains unbroken and the intruder is unable learn the encrypted security context sent. It can be seen that even though there is an authenticated state between the Security Token Service and the intruder service, the unbroken encryption prevents the intruder from learning the security context. A message sequence chart for a run between I, acting as A, and the Security Token Service service is shown in Fig. 6.8.

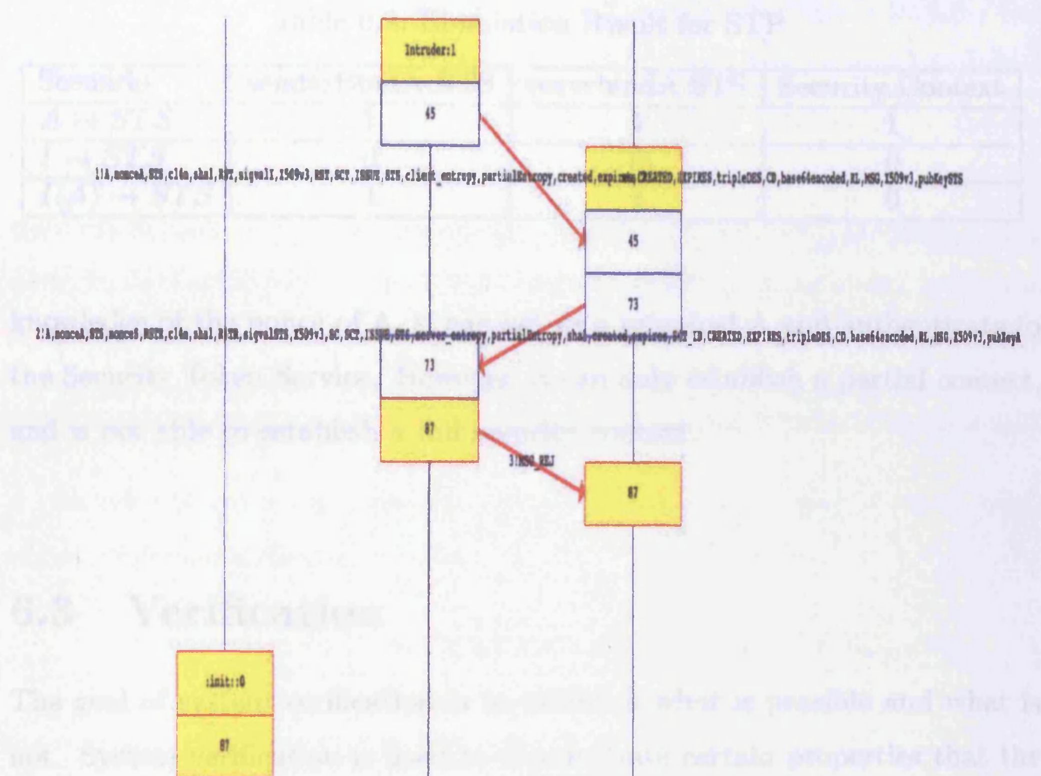


Figure 6.8: Message Sequence Chart for I(A) → STS.

The simulation results for Security Token Protocol are summarised in Table 6.2, which shows the three global variables ‘senderBindA_STS’, ‘recvrbindA_STS’ and ‘Security_Context’. When ‘senderBindA_STS’ and ‘recvrbindA_STS’ are set to 1, both Service A and the Security Token Service have authenticated to each other successfully. ‘Security_Context’ is set to 1 when a full agreement has been reached on a security context between Service A and the Security Token Service. It can be seen in I(A) → STS, that when the intruder has

Table 6.2: Simulation Result for STP

Scenario	senderBindA_STS	recvrbindA_STS	Security_Context
$A \rightarrow STS$	1	1	1
$I \rightarrow STS$	0	0	0
$I(A) \rightarrow STS$	1	1	0

knowledge of the nonce of A, it can act as a principal A and authenticate to the Security Token Service. However, it can only establish a partial context, and is not able to establish a full security context.

6.3 Verification

The goal of system verification is to establish what is possible and what is not. System verification is used to demonstrate certain properties that the model ought to possess. It is said that the system is "correct" when it satisfies all properties that obtain to it. A verification model not only deals with the behaviour of the system, but also its correctness requirements.

In practice, 'peer reviewing' and 'testing' are used as software verification techniques. A peer review is a scrutiny of software carried by software engineers without running the code. On the other hand, testing involves running the software. Peer reviewing and testing catch different errors at different cycles of development, and thus are often used together. These software verification approaches are used in a number of software projects.

When building a model for software verification it is tempting to build large complicated models which are as close to the problems as possible.

However, the most feasible approach to software verification is to keep the model in its simplest form, which represents the key attributes of the model for analysis. This approach helps in controlling the complexity of the model. An aim of model checking is to keep the model simple rather than building detailed models so that the essential features of the system are represented as a smallest sufficient model. The model can evolve if required. The type of abstraction for a model is dependant on the logical properties that are to be proved, and on the resource limits of the verification system.

In distributed system design, there are two main types of correctness claim: *Safety* and *Liveness*.

> "Safety is defined as the set of properties that the system may not violate, while liveliness is defined as the set of properties that the system must satisfy. Safety is concerned with the bad things that should be avoided, while liveliness defines the good things that capture the required functionality of the system" [Hol03].

The liveness properties of a system are modelled as Linear Temporal Logic formulas. The liveness properties of the Simple Message Exchange Protocol and the Security Token protocol, as modelled with Linear Temporal Logic formulas, has been presented previously, and in this chapter these properties are verified.

The working of *Spin* can be summarised as follows. Starting with a high-level verification model of a system, interactive and random simulation are performed using XSpin to check whether the model has the intended properties. A Promela correctness claim is generated from Linear Temporal Logic.

Spin is used to generate an on-the-fly optimised verification program from the high-level specification. This verification program is compiled and is executed to perform the verification. If any counterexamples of the correctness claims are detected, these can be fed back into the *Spin* simulator. The simulation trail can then be inspected in detail to determine the cause of any correctness violations.

Performing verification is an iterative process with increasingly detailed models. Each new model is verified under different assumptions about the environment and correctness properties. Selective data hiding can be used. When a verification run completes *Spin* provides hints on how to proceed, depending on the results. In the case of no hints, a clean run has been performed, that is, an exhaustive search that did not reveal any errors has been done.

To understand the verification results, it is necessary to be familiar with the output from the verification engine. The following terms which are relevant here. 'State vector' represents the size of a single state. 'Depth reached' represents the longest execution path. Error 0 means that the property was satisfied. If there are errors, these represent a violation of the Linear Temporal Logic property. 'State Stored' is the total number of states, i.e., the state space. These values can be used for comparison between the application of model checkers to Web services based security protocols.

Table 6.3: Verification Results for SMEP.

Scenario	State Size	Transitions
Satisfied	232 bytes	1659
Violated	252 bytes	6349

6.3.1 Simple Message Exchange Protocol

A Simple Message Exchange Protocol run is successful if Service A and Service B are able to bind to each other successfully in the presence of an active intruder. The properties of the Simple Message Exchange Protocol are expressed as Linear Temporal Logic formulas, and verification is performed using the following Linear Temporal Logic property. The verification found a Linear Temporal Logic property violation – the intruder was successfully able to exchange messages between services. The property was violated for 'I $\rightarrow$ B followed by A $\rightarrow$ I(B)' and 'A $\rightarrow$ I followed by I(A) $\rightarrow$ B'. This violation can be corrected by encrypting the nonces for all messages sent between services – this will prevent the intruder from using these values for further communication. The results are summarised in Table 6.3, which shows the state sizes for the model and the number of transitions it took to find the property violation. 'Transitions' represents the number of progressions *Spin* took to identify the violations in the Linear Temporal Logic property when it was satisfied or violated. 'State Size' shows the memory in bytes used for storing states.

Table 6.4: Verification Results for STP.

Scenario	State Size	Transitions
Satisfied	300 bytes	9250
Violated	300 bytes	4558

6.3.2 Security Token Protocol

A Security Token Protocol run is successful if an agreement can be reached on a security context between a requestor and a Security Token Service. These properties of the Security Token Protocol run are specified in the form of Linear Temporal Logic, as described below. It can be seen that the property is satisfied, i.e., Service A and the Security Token Service reach an agreement on a security context for 'A $\rightarrow$ STS'. It can also be seen that a security context is not established for 'I(A) $\rightarrow$ STS', where the Security Token Protocol is subjected to an attack from the intruder service, masquerading as Service A. The number of changes occurring is represented by the 'Transitions' made by *Spin* to recognise satisfied/violated Linear Temporal Logic properties in the model. The memory used to store all states during the run is recorded as 'State Size'.

The results obtained are summarised in Table 6.4. The table gives the number of bytes used for storing that state space and the transitions for both cases, where the LTL property is violated and when it is satisfied.

It can be seen that the Promela model represents the behaviour of the system. Simulation enables the detection of any anomaly in the behaviour of the model. In our case, a behavioural analysis is performed, and simulation provides the required results. The verification result for our model has also

been obtained.

6.4 Concluding Remarks

Simulations of the pushdown automaton model of the Simple Message Exchange Protocol have been carried out. The three steps of SMEP have been modelled as a pushdown automaton in Promela, as mentioned in Chapter 4. It can be seen from the simulation results that when these steps are executed successfully, Service A and Service B bind to each other, as shown in Table 6.1. A correct protocol run is accomplished on completion of the three steps resulting in Service A and Service B binding to each other. The behaviour of the SMEP pushdown automaton model is verified and results are summarised in Table 6.3. Representing the protocols using pushdown automata allows the behaviour of the protocol and the participating services to be modelled. On deviation from the steps, SMEP failed to bind successfully, as was shown by the values of the global variables.

Simulations for the Security Token Protocol were conducted, modelled as a three step pushdown automaton, as described earlier in Chapter 4. Each step of the protocol, when executed successfully, leads to a correct run of the STP protocol. Any variation in the steps leads to an incorrect run. An incorrect run means that a requesting service fails to bind to the STS or the service fails to agree on a security context. The pushdown automaton model for STP allows us to map the behaviour of the protocol and the participating services. The behavioural analysis of the STP is summarised in Tables 6.2 and 6.4.

The results for the simulation and verification for each scenario of the protocol runs for SMEP and STP may be summarised as follows. The simulation results showed the behaviour of the protocols in the active presence of an intruder. For each protocol scenario, the message sequence chart and the values of the global variables have been shown, in Table 6.1 for SMEP and in Table 6.2 for STP. The verification results are summarised in Tables 6.3 and 6.4 for SMEP and STP, respectively. It may be concluded that the Simple Message Exchange Protocol and the Security Token Protocol can be made more secure by encrypting the nonces. We suggest that all sensitive information be encrypted, and as little important information as possible should be left unencrypted. In the next chapter an extended intruder model, based on the Dolev-Yao model, for an XML Injection attack will be presented.

CHAPTER 7

XML Injection Attack Model

The Dolev-Yao threat model has been widely used in the past for analysis and verification of cryptographic protocols. Recently, the threat model has been adopted for the study and validation of Web services based cryptographic protocols. However, attack capabilities have increased over time resulting in new threats. The original Dolev-Yao model does not sufficiently address the potential of the attacker and the new threats that have arisen. To demonstrate the new behaviour of the intruder and the threat it introduces, the threat model must be extended. A wish-list has been suggested by Backes [BG05] on how to improve the model in accordance with Web services, but a model has not yet been produced. This chapter extends the Dolev-Yao model for Web services based cryptographic protocols by adding to the model an attacker capable of carrying out an XML injection attack.

A logic of an XML service can be influenced or undermined by XML injection, a type of command injection attack. In an XML injection attack[Con] some logic is inserted into a service to hinder the abilities of the service as defined by "The Web Application Security Consortium" and "Web Services Interoperability Organization". The insertion of XML content or XML structure into a document alters the intended rationale of the service. Furthermore, XML injection can cause the insertion of malicious content into the resulting message. These attacks can occur when user input is passed directly into an XML message stream. These attacks can be controlled/overcome by encrypting and/or signing parts of the document. If the content is injected into a signed XML document, it will be rejected by the service on verification.

This chapter builds an XML injection attack model in Promela, and runs this attack against the Simple Message Exchange Protocol and the Security Token Protocol. The purpose is to add to the capabilities of the Dolev-Yao intruder model and allow the intruder to simulate an XML injection attack. The integrity of the message is validated by checking if the message has been altered.

The XML injection attack model is built on the model presented in Chapter 5. The workings of the Simple Message Exchange Protocol and the Security Token Protocol are the same as described in Chapter 3.

7.1 Simple Message Exchange Protocol

The Simple Message Exchange Protocol is subjected to an XML Injection attack. 'Sender' and 'Receiver' are two legitimate services, and 'PI' is an intruder service. The Sender process sends a message to Receiver. The message is intercepted by the intruder process which has the choice of either altering the content or injecting a new element into the message. It sends the message off to the receiver service after the message has been altered. The Promela model for a Simple Message Exchange Protocol run under an XML injection attack is described as follows.

7.1.1 Types

This section describes the message types used in the Simple Message Exchange Protocol run. 'mtype' is used for defining symbolic names of numerical constants. There are six 'mtype' declarations for the Simple Message Exchange Protocol, as listed in Table 7.1. The 'General Purpose', 'XML Signature', 'XML Encryption', 'WS-Security' and 'TimeStamp' mtypes are similar to the ones explained earlier in Section 5.3.1. The 'XML Injection' category is added to the model. This consists of the 'AX' and 'INJ_INFO' mtypes. 'AX' represents alteration to the username token 'content' information in the message, and 'INJ_INFO' represents the 'injected' element in the Signature portion of the message.

Table 7.1: Simple Message Exchange Protocol Types

General Purpose	A, B, I, REQ, ACK, nonceA, nonceB, nonceI, ACCEPT, DECLINE
XML Signature	c14n, sha1, sigvalA, sigvalB, sigvalI, X509v3
XML Encryption	tripleDES, CD, base64encoded, RL
WS-Security	pubKeyA, pubKeyB, pubKeyI, privKeyA, privKeyB, privKeyI
XML Injection	AX, INJ_INFO
TimeStamp	CREATED, EXPIRES

7.1.2 Channels

Channels are used for communication between processes. Service A and Service B can communicate over two types of channels based on the type of message being sent. The first type, 'Msg1', represents the message {sender, sender_nonce, receiver, msg_type, TimeStamp, Signature, EncryptedData, pubKey}, which contains the username of the sender, a fresh nonce, the receiver username, the type of message (such as REQ), TimeStamp values, signature information, encrypted data and the public key of the receiver. The second type, 'Msg2', represents {sender, sender_nonce, receiver, receiver_nonce, msg_type, TimeStamp, Signature, EncryptedData, publicKey}. The message contains all fields present in Msg1, with the addition of receiver_nonce. Both channel 'chanONE' and 'chanTWO' are defined as non-buffered channels and are listed as follows:

```
chan chanONE = [0] of {mtype, mtype, mtype, mtype, TimeStamp, Signature,
                       EncryptedData, mtype};
chan chanTWO = [0] of {mtype, mtype, mtype, mtype, mtype, TimeStamp, Signature,
                       EncryptedData, mtype};
```

7.1.3 Global Variables

Global variables are used for formulating the Linear Temporal Logic formulas. These variables are similar to the ones defined in Section 5.3.1, 'SenderBindAB', 'SenderChallengeAB', 'RecvrChallengeAB' and 'RecvrbindAB'. 'SenderChallengeAB' and 'RecvrChallengeAB' are updated by Service A and Service B when initiating a run with each other. When both Service A and Service B have authenticated successfully, they bind to each other, and 'SenderBindAB' and 'RecvrBindAB' are set to 1. The Linear Temporal Logic property for this model is the same as defined in previous chapters:

[] (([] !SenderBindAB) || (!SenderBindAB U RecvrChallengeAB))

[] (([] !RecvrBindAB) || (!RecvrBindAB U SenderChallengeAB))

The 'typedef' structures used in the Simple Message Exchange Protocol consist of the XML based TimeStamp, Signature and Encryption information for the protocol. All the complex structures indicating the injected element information and the XML Signature structure are given in Table 7.2.

7.1.4 Principal Processes

There are three principals for the Simple Message Exchange Protocol: a 'Sender' process representing Service A, a 'Receiver' process representing Service B, and 'PI' symbolising the intruder service. Each service is represented as a Promela processes. Service A is modelled as process 'SenderA', service B is presented as process 'ReceiverB', and the intruder service is defined as process 'PI'. These services communicate with each other over shared channels.

Table 7.2: Simple Message Exchange Protocol Global Variables.

TimeStamp	typedef TimeStamp{ mtype Created; mtype Expires; }; TimeStamp ts[6];
XML Signature	typedef Signature { mtype CanoncalizationMethod ; mtype SignatureMethod ; mtype Reference ; mtype SignatureValue ; mtype KeyInfo; }; Signature sig[6];
Signature Injection	typedef typedef SignatureInjection{ mtype CanoncalizationMethod ; mtype SignatureMethod ; mtype Reference ; mtype SignatureValue ; mtype KeyInfo; mtype InjectedInfo; }; SignatureInjection iSig[1];
XML Encryption - Encrypted Data	typedef EncryptedData{ mtype EncryptionMethod ; mtype CipherData ; mtype CipherValue ; mtype ReferenceList ; mtype DataReference ; mtype KeyInfo; }; EncryptedData edata[6];
XML Encryption - Encrypted Key	typedef EncryptedKey { mtype EncryptionMethod ; mtype CipherData ; mtype CipherValue ; mtype ReferenceList ; mtype DataReference ; mtype KeyInfo; }; EncryptedKey ek[6];

Init Process The 'init' process contains all the information that needs to be instantiated at the beginning of the run. This process contains other processes or statements. The 'init' process is defined as follows:

```
init {
    run SenderA(A, B, nonceA);
    run ReceiverB(B, nonceB);
    run PI(I, nonceI)
}
```

All three processes are instantiated at the beginning of the execution with their respective initialisation parameters. 'SenderA' is initialised with its identity A, random nonce value nonceA, and the receiver process it wants to communicate with, B. The 'ReceiverB' process is initialised with its identity B and a random nonce value nonceB. The receiver process is not an initiator process, it can only receive messages and reply. 'PI', the intruder process, is initialised with its identity I and its random nonce I.

Sender Process The Sender Process initiates a three-step message exchange with other communicating services. In the first step, the sender populates its local and global variables for Signature, Encryption, and TimeStamp, and assigns the senderChallengeAB global variable by calling SenderChallenge(me, recvr), and then sends the message on 'chanONE'. The message contains {A, nonceA/nonceI, B/I, REQ, TimeStamp, Signature, EncryptedData, pubKeyB/pubKeyI}. In the second step, the sender waits for a response from the receiver service on 'chanTWO' of the form {B/I, nonceB/nonceI, A, nonceA, ACK/DECLINE, TimeStamp, Signature, Encrypted-

Data, pubKeyA}. It decrypts the message, verifies TimeStamp, validates Signature and binds to the service. In the third step, if the run is successful, the sender sends an acknowledgement back to the receiver, {A, nonceA, B/I, nonceB/I, ACCEPT, TimeStamp, Signature, EncryptedData, pubKey-B/pubKeyI }, or else it does not return any message back to the receiver. The sender process is similar to the one in Section 5.3.1. The 'SenderA' process is defined as follows.

```
proctype SenderA(mtype me; mtype recvr; mtype my_nonce)
{
    atomic {
        senderchallenge(me, recvr);
        pubKey = pubKeyB;
        chanONE ! me,my_nonce,recvr,REQ,ts[0],sig[0],edata[0],pubKey;
    }
    atomic {
        chanTWO?recvr,recvr_nonce,me,my_nonce,msg_type,ts[3], sig[3], edata[3],pubKey;
        Decryption(pubKey, privateKey);
        VerifyFreshness(ts[3].Created,ts[3].Expires);
        VerifySignature(sig[3].CanoncalizationMethod, sig[3].SignatureMethod,
            sig[3].Reference, sig[3].SignatureValue, sig[3].KeyInfo);
        senderbind(me,recvr);
        if
        :: (msg_type == ACK) → sig[0].Reference = ACCEPT;
            chanTWO!me,my_nonce,recvr,recvr_nonce,ACCEPT,ts[0],sig[0],edata[0], pubKey;
        :: else skip
        fi;
    }
}
```

Receiver Process The receiver process is always waiting for incoming messages. 'Receiver' is a three-step process. First, it waits for messages from services on channel 'chanONE'. On receiving a message of type {A/I, nonceA/nonceI, B, REQ, TimeStamp, Signature, EncryptedData, pubKeyB} it symbolically decrypts the message with its private key, verifies the freshness of the message, validates the signature, and updates the global variable RecvrChallengeAB by calling recvrchallenge(sender, me). Secondly, the receiver creates a response message to send back to the service. It populates its local and global database of variables, representing Encryption, Signature, and TimeStamp information, and sends a response of type {B, nonceB, A/I, nonceA/nonceI, ACK/DECLINE, TimeStamp, Signature, EncryptedData, pubkeyA/pubKeyI} on 'chanTWO'. Thirdly, the receiver goes into a waiting state until a response from the initiator process is received on 'chanTWO'. Service B receives the message of the form {A/I, nonceA/nonceI, B, nonceB, ACCEPT, TimeStamp, Signature, EnncryptedData, pubkeyB}. The service decrypts the message, verifies the TimeStamp and the Signature information, and then updates the Linear Temporal Logic global variables by invoking recvrbind(sender, me). The 'ReceiverB' process is listed as follows.

```
proctype ReceiverB(mtype me; mtype my_nonce)
{
    atomic {
        chanONE?sender,sender_nonce,eval(me),msg_type,ts[1],sig[1], edata[1],pubKey;
        atomic{
            Decryption(pubKey, privateKey);
            VerifyFreshness(ts[1].Created,ts[1].Expires);
            VerifySignature(sig[1].CanoncalizationMethod, sig[1].SignatureMethod,
                sig[1].Reference, sig[1].SignatureValue, sig[1].KeyInfo);
            recvrchallenge(sender,me);
```

```
            pubKey = pubKeyA;
            chanTWO!me,my_nonce,sender,sender_nonce,msg_type,ts[2],sig[2],edata[2],pubKey;
        }
    }
    atomic {
        chanTWO?eval(sender),eval(sender_nonce),eval(me),eval(my_nonce),msg_type,
            ts[4],sig[4],edata[4],pubKey;
        Decryption(pubKey, privateKey);
        VerifyFreshness(ts[4].Created,ts[4].Expires);
        VerifySignature(sig[4].CanoncalizationMethod, sig[4].SignatureMethod,
            sig[4].Reference, sig[4].SignatureValue, sig[4].KeyInfo);
        recvrbind(sender,me);
    }
}
```

Intruder Process The intruder is an extension of the Dolev-Yao model with the capability of an XML Injection attack. The intruder has the ability to inject content into the message, or it can inject an element into the Signature element. It listens on 'chanONE' for any messages between Service A and Service B. When a message is sent from Service A to Service B, it intercepts the message. It either injects 'content' or 'element' into the message. Altering the message 'content' or 'element' causes the message to be rejected by Service B. When injecting 'element' into the message, 'iSig' of type 'SignatureInjection' is sent with the same element contents as well as 'InjectedInfo'. The intruder can alter the 'content', for example, by changing the username in the message by adding 'X' to the username to cause authentication to fail. The intruder can replay two types of message, {A, nonceA, B, REQ, TimeStamp, SignatureInjection, EncryptedData, pubKey} representing an 'element' alteration, and {AX, nonceA, B, REQ, TimeStamp, Signature, EncryptedData, pubkeyB} representing a 'content' alteration to

the username token.

```
proctype PI(mtype me; mtype my_nonce)
{
    do
    ::atomic {
        chanONE?sender,sender_nonce,recvr,msg_type,ts[5],sig[5],edata[5],pubKey;
        sender = A;
        chanONE!sender,sender_nonce,recvr,REQ,ts[5],iSig[0],edata[5],pubKey;
    }
    ::atomic {
        chanONE?sender,sender_nonce,recvr,msg_type,ts[5],sig[5],edata[5],pubKey;
        sender = AX;
        chanONE!sender,sender_nonce,recvr,REQ,ts[5],sig[5],edata[5],pubKey;
    }
    od
}
```

So far the building blocks have been presented for a Simple Message Exchange Protocol in terms of Promela constructs for an XML injection attack.

7.2 Security Token Protocol

In this section the Security Token Protocol is subjected to an XML injection attack. The model consists of a 'Sender', 'STS' (representing the Security Token Service), and an 'Intruder' process. The sender and the Security Token Service process behave in the same manner as described in Chapter 5. The intruder process intercepts the request for a security token from the sender service. The intruder is capable of changing 'content' in the Username Token

Table 7.3: Security Token Protocol Types

General Purpose	A, B, I, REQ, ACK, nonceA, nonceB, nonceI, ACCEPT, DECLINE
XML Signature	c14n, sha1, sigvalA, sigvalB, sigvalI, X509v3
XML Encryption	tripleDES, CD, base64encoded, RL
WS-Security	pubKeyA, pubKeyB, pubKeyI, privKeyA, privKeyB, privKeyI
XML Injection	AX, INJ_INFO, SCTX, ISSUEX
TimeStamp	CREATED, EXPIRES

and Request Security Token (RST) or it can add 'element' to the RST. The message is then sent to the Security Token Service for processing. The alterations to the message can result in authentication failures or security context agreement failures. The Promela version of the model is listed below.

7.2.1 Types

Table 7.3 lists all the message types used in the model. In the table there are two more 'mtype' entries in addition to the ones defined in Section 5.3.2. 'AX' is used to alter content information in the XML message. 'INJ_INFO' represents the injected element in the Signature portion of the message. The intruder is allowed the capability to alter the RST. The intruder can alter the 'TokenType' and 'RequestType' elements in the issuance request to 'SCTX' and 'ISSUEX' respectively.

7.2.2 Channels

The Security Token Protocol uses three channels that are similar to those described in Section 5.3.2. The channels are identified by the type of message they can support. There are three types of messages in the Security Token Protocol: 'Msgl' is of the form {UsernameToken, receiver, Signature, RST, RequestSecurityToken, TimeStamp, EncryptedData, pubKey}; 'Msg2' is of the form {UsernameToken, UsernameToken, RSTR, Siganture, SC, Request-SecurityTokenResponse, TimeStamp, EncryptedData, pubKey}; and 'Msg3' is of the form {SC/MSG_REJ}.

```
chan rstChan = [0] of {UsernameToken, mtype, Signature, mtype, RequestSecurityToken,
                       TimeStamp, EncryptedData, mtype};
chan chanTWO = [0] of {UsernameToken, UsernameToken, mtype, Signature, mtype,
                       RequestSecurityTokenResponse, TimeStamp, EncryptedData, mtype};
chan ackChan = [0] of {mtype};
```

7.2.3 Global Variables

The global variables used are similar to those defined in Section 5.3.2. There are six basic global variables used for the Linear Temporal Logic formulas: 'SenderBindA_S', 'SenderChallengeA_S', 'RecvrChallengeA_S', 'RecvrbindA_S', 'partial_SC' and 'security_context'. The Linear Temporal Logic formula used for verification is:

```
[] ( ([] !senderbindAS) || (!senderbindAS U recvrchallengeAS) )

[] ( ([] !recvrbindAS) || (!recvrbindAS U senderchallengeAS) )
```

[] (([] !securitycontext) || (!securitycontext U partialSC))

Tables 7.4 and 7.5 contain the complex structures for 'TimeStamp', 'Signature'. 'EncryptedData', 'EncryptedKey', 'UsernameToken', 'RequestSecurityToken', 'RequestSecurityTokenResponse' and 'Injected RequestSecurityToken'. The 'iRequestSecurityToken' structure is used by the intruder when the message is injected with an XML Element. It is similar to the 'RequestSecurityToken' structure but with the addition of an 'InjectedInfo' element. 'InjectedInfo' represents any element data introduced to the request security token structure. The value of the 'InjectedInfo' is set to 'INJ_INFO'.

7.2.4 Principal Processes

The principal processes for the Security Token Protocol under XML Injection attack are defined as follows.

Init Process: The 'init' process instantiates all three processes with their respective data values. The 'Intruder' is initialised with its identity I, the username of the service it will communicate with (i.e., the Security Token Service), and its nonce, nonceI. The 'Sender' process is initialised with its identity A, the identity of the Security Token Service to which it will send the request for a security token, and its fresh nonce, nonceA. The 'STS' is a security token issuing process. It is initialised with its identity STS and nonceS.

```
init {
    run Intruder(I, STS, nonceI);
    run Sender(A, STS, nonceA);
```

Table 7.4: Security Token Protocol Global Variables.

TimeStamp	typedef TimeStamp{ mtype Created; mtype Expires; }; TimeStamp ts[6];
XML Signature	typedef Signature { mtype CanoncalizationMethod ; mtype SignatureMethod ; mtype Reference ; mtype SignatureValue ; mtype KeyInfo; }; Signature sig[6];
XML Encryption - Encrypted Data	typedef EncryptedData{ mtype EncryptionMethod ; mtype CipherData ; mtype CipherValue ; mtype ReferenceList ; mtype DataReference ; mtype KeyInfo; }; EncryptedData edata[6];
XML Encryption - Encrypted Key	typedef EncryptedKey { mtype EncryptionMethod ; mtype CipherData ; mtype CipherValue ; mtype ReferenceList ; mtype DataReference ; mtype KeyInfo; }; EncryptedKey ek[6];
UsernameToken	typedef UsernameToken { mtype Username; mtype Nonce; }; UsernameToken ust[7];

Table 7.5: Security Token Protocol Global Variables.

<table>
<tr><td>RequestSecurityToken</td><td>typedef RequestSecurityToken{
mtype TokenType ;
mtype RequestType ;
mtype AppliesTo;
mtype Entropy;
mtype EntropicMode;
mtype rst_created;
mtype rst_expires;
}; RequestSecurityToken rst[2];</td></tr>
<tr><td>Injected RequestSecurityToken</td><td>typedef iRequestSecurityToken{
mtype TokenType ;
mtype RequestType ;
mtype AppliesTo;
mtype Entropy;
mtype EntropicMode;
mtype rst_created;
mtype rst_expires;
mtype InjectedInfo
};iRequestSecurityToken irst[1];</td></tr>
<tr><td>RequestSecurityToken</td><td>typedef RequestSecurityTokenResponse{
mtype TokenType ;
mtype RequestType ;
mtype AppliesTo;
mtype Entropy;
mtype EntropicMode;
mtype ComputedKey;
mtype rstr_created;
mtype rstr_expires;
mtype sct_id;
}; RequestSecurityTokenResponse rs[2];</td></tr>
</table>

```
    run sts(STS, nonceS);
}
```

Sender Process: The sender process performs a three-step message run between itself and the Security Token Service. First, it populates the 'typedef' structures with its Encryption, Signature, TimeStamp and Request for Security Token values. It invokes 'SenderChallenge(me, recvr)' and sends the message {UsernameToken(A, nonceA), STS, Signature, RST, RequestSecurityToken, TimeStamp, EncryptedData, pubKeySTS} on channel 'rstChan'. Secondly, it waits for the message {UsernameToken(STS, nonceS), UsernameToken(A, nonceA), RSTR, Signature, SC, RequestSecurityTokenResponse, TimeStamp, EncryptedData, pubKeyA} from the Security Token Service on 'rstrChan'. The message is decrypted, the timestamp is verified, the signature and authentication information is validated, and the sender binds to the Security Token Service and agrees on a security context after processing the security context information received from the Security Token Service. Thirdly, the sender process sends an acknowledgement response message, {SC/MSG_REJ}, to the security token service on channel 'ackChan'. The Sender process is listed as follows:

```
proctype Sender(mtype me; mtype recvr; mtype my_nonce)
{
    atomic {
        SenderChallenge(me, recvr);
        rstChan!ust[0],recvr,sig[0],RST,rst[0],ts[0],edata[0],pubKey ;
    }
    atomic {
        rstrChan?ust[4],ust[3],rstr,sig[3],SC,rs[1],ts[3],edata[3],pubKey;
        Decryption(pubKey, privateKey);
```

```
        VerifyFreshness(ts[3].Created,ts[3].Expires);
        VerifySignature(sig[3].CanoncalizationMethod, sig[3].SignatureMethod,
            sig[3].Reference, sig[3].SignatureValue, sig[3].KeyInfo);
        AuthenticateResponse(rstr);
        SenderBind(ust[4].Username, ust[3].Username);
        SCAgreement(rs[1].AppliesTo, rs[1].TokenType, rs[1].RequestType,
            rs[1].EntropicMode, rst[0].Entropy, rs[1].Entropy,rs[1].ComputedKey,
            rs[1].rstr_expires, ust[3].Nonce, ust[3].Username, ust[4].Username, rs[1].sct_id);
    }
    if
        :: (Security_Context == FLAG) → msg = SC
        :: (Security_Context != FLAG) → msg = MSG_REJ
    fi;
    ackChan!msg
}
```

Receiver Process: The receiver process is the Security Token Service listening for requests for tokens on 'rstChan'. First, the Security Token Service awaits a the request message of the form {UsernameToken(A, nonceA), STS, Signature, RST, RequestSecurityToken, TimeStamp, EncryptedData, pubKeySTS}. On receiving the message the Security Token Service symbolically decrypts it with its private key, verifies the freshness, validates the signature and authenticates the request, invokes 'RecvrChallenge(sender, receiver)', and finally agrees on a partial security context. Secondly, the Security Token Service populates its RequestSecurityResponse complex data structure with the appropriate requested token information and sends the response on 'rstrChan'. The response message contains {UsernameToken(A, nonceA), UsernameToken(STS, nonceS), RSTR, Signature, SC, RequestSecurityTokenResponse, TimeStamp, EncryptedData, publicKeyA}. If the Security Token Service fails to authenticate, or is unable to reach an agreement on

a partial security context, it sends a request security token response token with NULL fields but returns the original values in the request back to the requester. Thirdly, the Security Token Service waits for an acknowledgement from the requestor service either as REJ or SC on 'ackChan'. The process is represented as follows:

```
proctype sts(mtype me; mtype my_nonce)
{
    atomic {
        rstChan?ust[1],eval(me),sig[1],msg_type,rst[1],ts[1],edata[1],pubKey;
        Decryption(pubKey, privateKey);
        VerifyFreshness(ts[1].Created,ts[1].Expires);
        VerifySignature(sig[1].CanoncalizationMethod, sig[1].SignatureMethod,
            sig[1].Reference, sig[1].SignatureValue, sig[1].KeyInfo);
        RecvrChallenge(ust[1].Username, me);
        AuthenticateRequest(msg_type);
        RecvrBind(ust[1].Username, me);
        PartialSCAgreement(rst[1].AppliesTo, rst[1].TokenType, rst[1].RequestType,
            rst[1].EntropicMode, rst[1].Entropy);
        rstrChan!ust[1],ust[2],RSTR,sig[2],SC,rs[0],ts[0],edata[2],pubKey;
    }
    ackChan?x;
}
```

Intruder Process: The intruder process is always active. It listens for messages on channel 'rstChan', and intercepts and alters them. The intruder on start-up populates its local variables and 'typedef' structures. The intruder can alter messages by 'content' injection or by 'element' injection to the message. It inserts 'content' into the UsernameToken by adding 'X' to the username, where 'X' represents any arbitrary data. The intruder sends

the message off on 'rstChan' to the Security Token Service. This alteration to the requestor's username causes failure in authentication. When the intruder alters the content of the request for a security token, it adds 'X' to 'RequestType' or to 'TokenType', thus causing a failure in establishment of a partial context or a full security context. The intruder's other capability is to inject an 'element' into the request security token message structure and send the message forward to the Security Token Service. The intruder process is represented as follows:

```
proctype Intruder(mtype me; mtype recvr; mtype my_nonce)
{
do
    :: atomic{
        rstChan?ust[0],recvr,sig[4],RST,rst[2],ts[0],edata[4],pubKey;
        ust[0].Username = AX;
        rstChan!ust[0],recvr,sig[4],RST,rst[2],ts[0],edata[4],pubKey;
    }
    :: atomic{
        rstChan?ust[0],recvr,sig[4],RST,rst[2],ts[0],edata[4],pubKey;
        rst[2].TokenType = SCTX;
        rst[2].RequestType = ISSUEX;
        rstChan!ust[0],recvr,sig[4],RST,rst[2],ts[0],edata[4],pubKey;
    }
    :: atomic{
        rstChan?ust[0],recvr,sig[4],RST,rst[2],ts[0],edata[4],pubKey;
        rstChan!ust[0],recvr,sig[4],RST,irst[0],ts[0],edata[4],pubKey;
    }
od
}
```

7.3 Concluding Remarks

In this chapter a model to expose the Simple Message Exchange Protocol and the Security Token Protocol to an XML injection attack has been built. The basic definitions of the legitimate user involved in the protocol run are similar to those in Chapter 5. The intruder model of Dolev-Yao is extended to encompass XML injection attacks. There are two ways the intruder attacks the messages in the protocol run – either by altering the contents of the elements or by adding an element to the message. In the next chapter, simulations of the Simple Message Exchange Protocol and Security Token Protocol models for analysing behaviour are run.

CHAPTER 8

Simulation and Verification for XML Injection Attack

This chapter presents the results of the simulation and verification of both the Simple Message Exchange Protocol and the Security Token Protocol under an XML injection attack. The simulation results are presented in the form of snapshots of message sequence charts and the global variables output. Verification is performed for both the protocols, which are checked for the same Linear Temporal Logic properties that were described in Section 3.7.

8.1 Simulation Results

8.1.1 Simple Message Exchange Protocol

The simulation results for the Simple Message Exchange Protocol will now be given. The protocol is subjected to two different types of XML injection attack. The first type of attack alters the 'content' of the element in the

message, and the second type of attack adds an element to the element structure represented by complex 'typedef' structures. Thus, two simulation scenarios are presented. In the first XML content is injected, i.e., the values are the elements are modified in some way. In the second an element is added to one of the WS-Security structures.

XML Content Injection

Service A sends a message to Service B. The message is intercepted by the intruder who is listening on the same channel, 'chanONE'. The intruder adds an 'X' to the username and sends it off to Service B. 'X' represents any arbitrary piece of data that can be added to the content of the element. Service B receives the message and tries to authenticate with A but fails to do so because of the alteration to the UsernameToken of the requesting service, A, and sends the message back to Service A.

It can be seen from the global variables that services A and B fail to bind with each other. Service A updates the global variable 'senderchallengeAB' to 1 before sending the message on 'chanONE'. The message is intercepted by the intruder service, which adds to the content of the username token, and sends it to Service B. Service B starts its authentication process by trying to update 'recvrchallengeAB' but is unable to update it to 1. This leads to the end of the simulation and the end of the protocol run.

Figure 8.1 represents a snapshot of a message sequence chart when content is altered in the message stream. The values in the yellow boxes represent the processes in the Simple Message Exchange Protocol model. 'init::0' in the first column represents the 'init' process which initialises all the func-

tions. 'SenderA:1' in the second column represents the Sender process. 'ReceiverB:3' in the third column represents the receiver process, and 'PI:2' in the fourth column represents the intruder process. SenderA sends a message to ReceiverB containing {A, nonceA, B, REQ, ts[0], sig[0], edata[0], pubKeyB}.The values for TimeStamp, XML Signature and XML Encryption for SenderA are described using 'ts[0]', 'sig[0]' and 'edata[0]'. 'A', 'nonceA' and 'B' are the unencrypted information in the message. The message is intercepted by the intruder, which alters the username 'content' of the message to {AX, nonceA, B, REQ, ts[0], sig[0], edata[0], pubKeyB}, and sends the message to the original intended recipient, B. ReceiverB receives and processes the message. The alteration to the username of SenderA causes the message to be rejected by ReceiverB. ReceiverB sends the following message {B, nonceB, AX, nonceA, DECLINE, ts[2], sig[2], edata[2], pubKeyA}. The message contain the ReceiverB values for TimeStamp, XML Signature and XML Encryption represented by 'ts[2]', 'sig[2]' and 'edata[2]', respectively. The protocol run terminates after the message is received by SenderA.

XML Element Injection

Service A sends a message to Service B on 'chanONE'. The intruder, as always, is listening on the channel. It intercepts the message and adds an element 'INJ_INFO' to the Signature element. The intruder then forwards the message to Service B. Service B processes the message. It successfully authenticates the message and sends an 'ACCEPT' to Service A.

It can be seen that the global authentication variables are updated successfully for authentication. Service A binds to Service B, and Service B

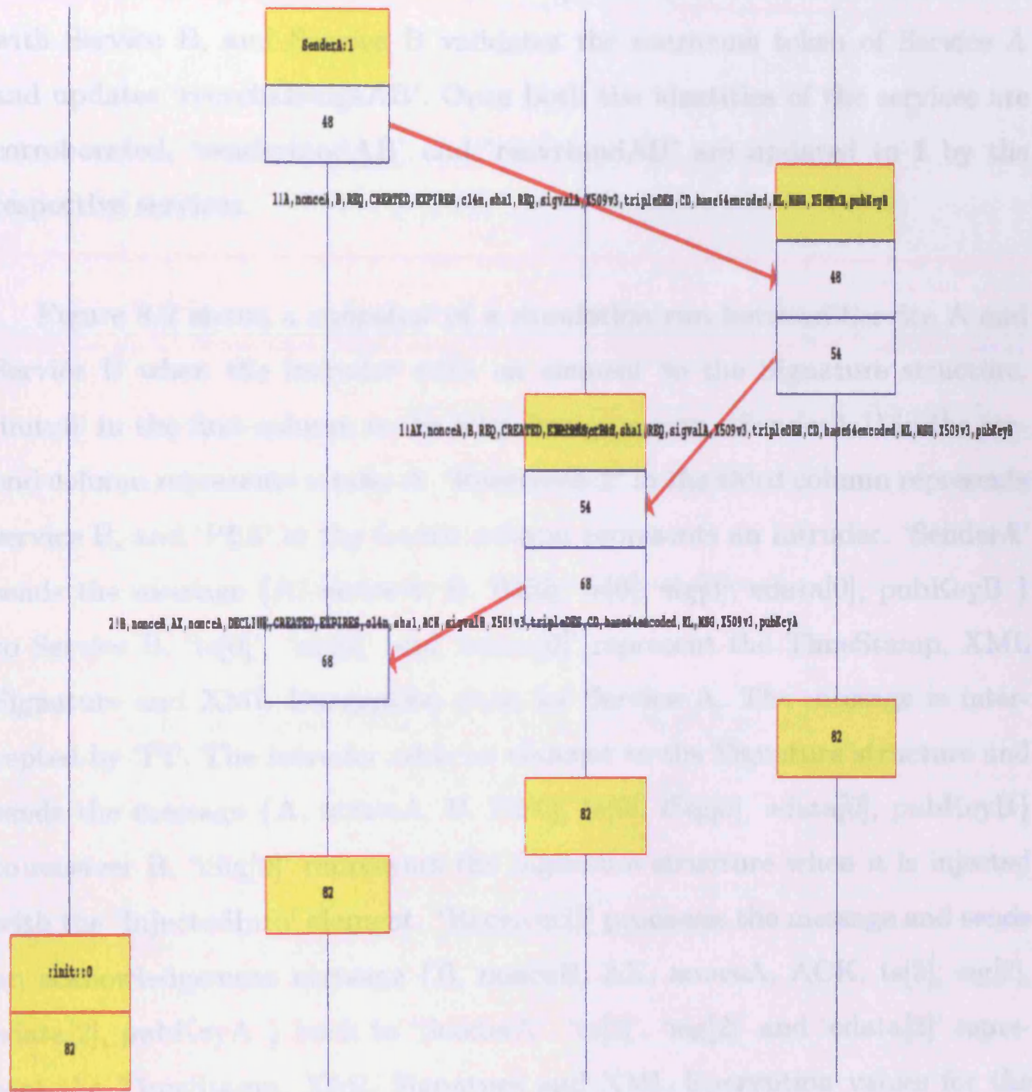


Figure 8.1: Message Sequence Chart for SMEP XML Content Injection Attack.

binds to service A without being aware that an extra element has been injected. Service A updates 'senderchallengeAB' to 1 when it initiates a run with Service B, and Service B validates the username token of Service A and updates 'recvchallengeAB'. Once both the identities of the services are corroborated, 'senderbindAB' and 'recvrbindAB' are updated to 1 by the respective services.

Figure 8.2 shows a snapshot of a simulation run between Service A and Service B when the intruder adds an element to the Signature structure. 'init::0' in the first column is the initialising process. 'SenderA:1' in the second column represents sender A. 'ReceiverB:2' in the third column represents service B, and 'PI:3' in the fourth column represents an intruder. 'SenderA' sends the message {A, nonceA, B, REQ, ts[0], sig[0], edata[0], pubKeyB } to Service B. 'ts[0]', 'sig[0]' and 'edata[0]' represent the TimeStamp, XML Signature and XML Encryption data for Service A. The message is intercepted by 'PI'. The intruder adds an element to the Signature structure and sends the message {A, nonceA, B, REQ, ts[0], iSig[0], edata[0], pubKeyB} to receiver B. 'iSig[0]' represents the Signature structure when it is injected with the 'InjectedInfo' element. 'ReceiverB' processes the message and sends an acknowledgement message {B, nonceB, AX, nonceA, ACK, ts[2], sig[2], edata[2], pubKeyA } back to 'SenderA'. 'ts[2]', 'sig[2]' and 'edata[2]' represent the TimeStamp, XML Signature and XML Encryption values for the receiver, B. 'ReceiverB' fails to identify the presence of an element injection attack. 'SenderA' receives the message, processes it and sends a reply {A, nonceA, B, nonceB, ACCEPT, ts[0], sig[0], edata[0], pubKeyB} to Service B.

The simulation results for the Simple Message Exchange Protocol are

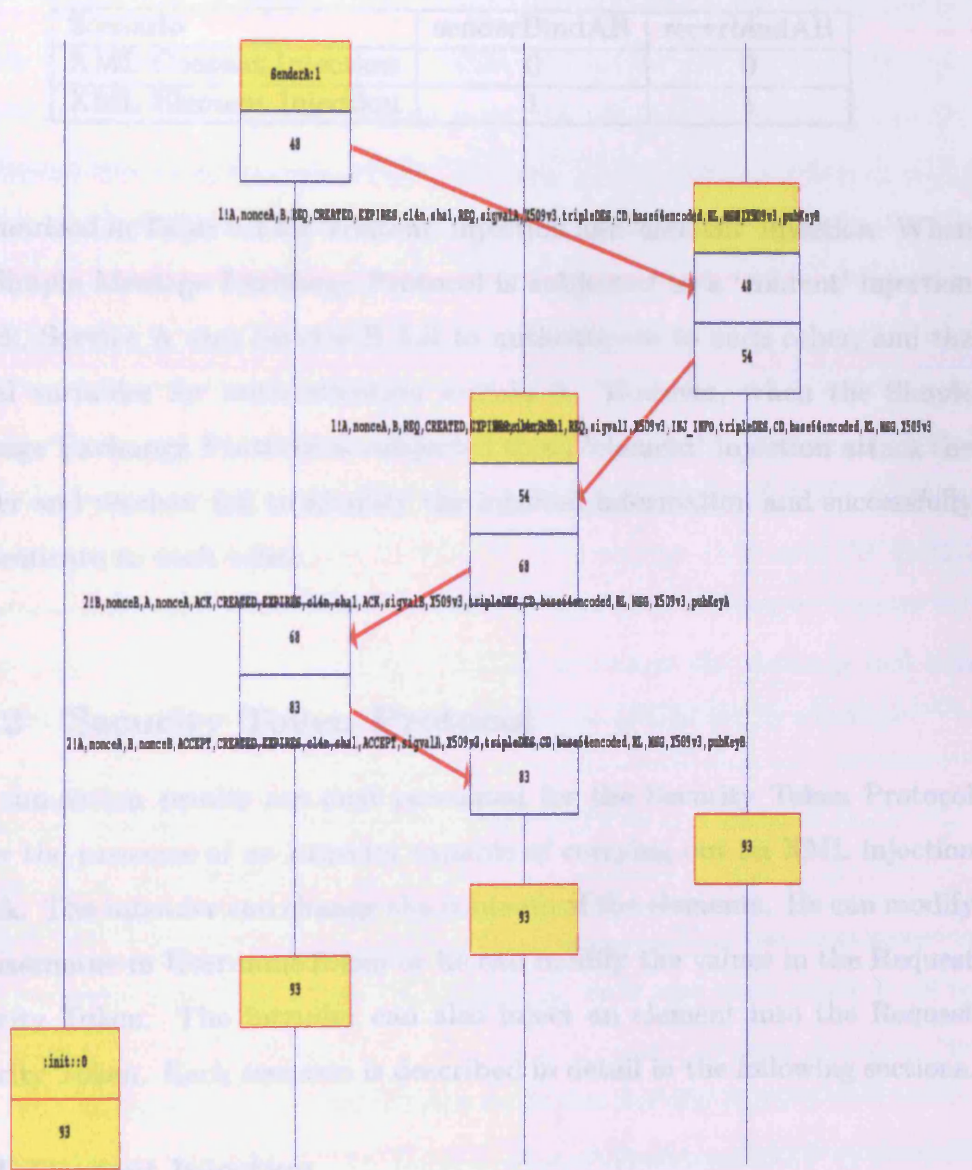


Figure 8.2: Message Sequence Chart - SMEP XML Element Injection Attack.

Table 8.1: Simulation Result for SMEP

Scenario	senderBindAB	recvrbindAB
XML Content Injection	0	0
XML Element Injection	1	1

summarized in Table 8.1 for 'content' injection and 'element' injection. When the Simple Message Exchange Protocol is subjected to a 'content' injection attack, Service A and Service B fail to authenticate to each other, and the global variables for authentication remain 0. However, when the Simple Message Exchange Protocol is subjected to an 'element' injection attack the sender and receiver fail to identify the injected information and successfully authenticate to each other.

8.1.2 Security Token Protocol

The simulation results are next presented for the Security Token Protocol under the presence of an intruder capable of carrying out an XML injection attack. The intruder can change the contents of the elements. He can modify the username in UsernameToken or he can modify the values in the Request Security Token. The intruder can also inject an element into the Request Security Token. Each scenario is described in detail in the following sections.

XML Content Injection

Username Token Injection Figure 8.3 shows a snapshot for an XML content injection attack. 'init', 'Sender', 'STS' and 'Intruder' are the four processes in the model. The numbers represent the sequence of steps in the simulation run. In step 48 'Sender' directs a message {ust[0], STS, sig[0], RST,

rst[0], ts[0], edata[0], pubKeySTS} to the Security Token Service. 'ust[0]', 'STS', 'sig[0]', 'RST', 'rst[0]', 'ts[0]' and 'edata[0]' represent the Username-Token, the receiver username, the signature information of the sender, the type of message, the request security token information, TimeStamp and Encryption information for the 'Sender' process. The message is intercepted by the intruder service. The intruder service only changes the username content of Service A and sends the message {ust[0], recvr, sig[4], RST, rst[2], ts[0], edata[4], pubKey} to the Security Token Service, as shown in execution step 50. The Signature, Encryption, Request Security Token and timestamp information remains unchanged. The 'STS' process receives the message and sends the response {ust[1], ust[2], msg'type, sig[2], SC, rs[0], ts[0], edata[2], pubKey} to the Sender process in step 77. The response contains the partial security context and information on how to establish a security context between 'AX' and the 'STS' service. 'Sender' processes the message and fails to authenticate its username token, and sends a {MSG_REJ} message.

Service A sends a request for a security token to the Security Token Service provider, the 'STS' process. The intruder listens for the request on the channel 'rstChan'. It intercepts this message and modifies the 'Username' in the Username Token structure by adding 'X' to it and then sends it off to the Security Token Service, 'STS'. The Security Token Service processes the message and fails to bind with Service A. When Service A initiates a protocol run with the Security Token Service 'SenderChallengeA_STS' is updated to 1. On the receiver end, the Security Token Service fails to validate the 'AX' username and does not update 'RecvrChallengeA_STS' to 1. This leads to the failure of authentication between Service A and the Security Token Service. The Security Token Service processes the rest of the message and

sends a security context token back to Service A. Service A rejects this token as the token contains the username 'AX'.

RST Injection Figure 8.4 shows a snapshot of a message sequence exchange chart for Service A and the Security Token Service in the presence of a content injection attack on values in the request security token structure. Service A sends a request {ust[0], recvr, sig[0], RST, rst[0], ts[0], edata[0], pubKeySTS} to 'STS' in execution step 48. The message is intercepted by the intruder, which alters the contents of the request security token structure. The intruder is only allowed to tamper with TokenType and RequestType. It then sends the message {ust[0], recvr, sig[4], RST, rst[2], ts[0], edata[4], pubKey} back on the channel to the Security Token Service in execution step 51. The Security Token Service is able to authenticate the sender, but is unable to agree on a partial context with Service A. In step 79 it sends a response {ust[1] , ust[2] , RSTR, sig[2], SC, rs[0], ts[0], edata[2], pubKeyA} with all NULL values back to the requestor, Service A. Service A, upon processing the request, is unable to establish a security context and sends a {MSG_REJ} response.

The global variables 'SenderChallengeA_STS', 'RecvrChallengeA_STS', 'SenderBindA_STS' and 'RecvrBindA_STS' are updated accordingly to 1. But both the context establishment variables, 'partial_SC' and 'Security_Context', fail. The Security Token Service fails to accept 'TokenType' and 'RequestType' in the request to the Security Token Service. It sends the request security token response back to Service A, containing 'NULL' fields, thus leading to a failure in establishing a security context with a valid Service A.

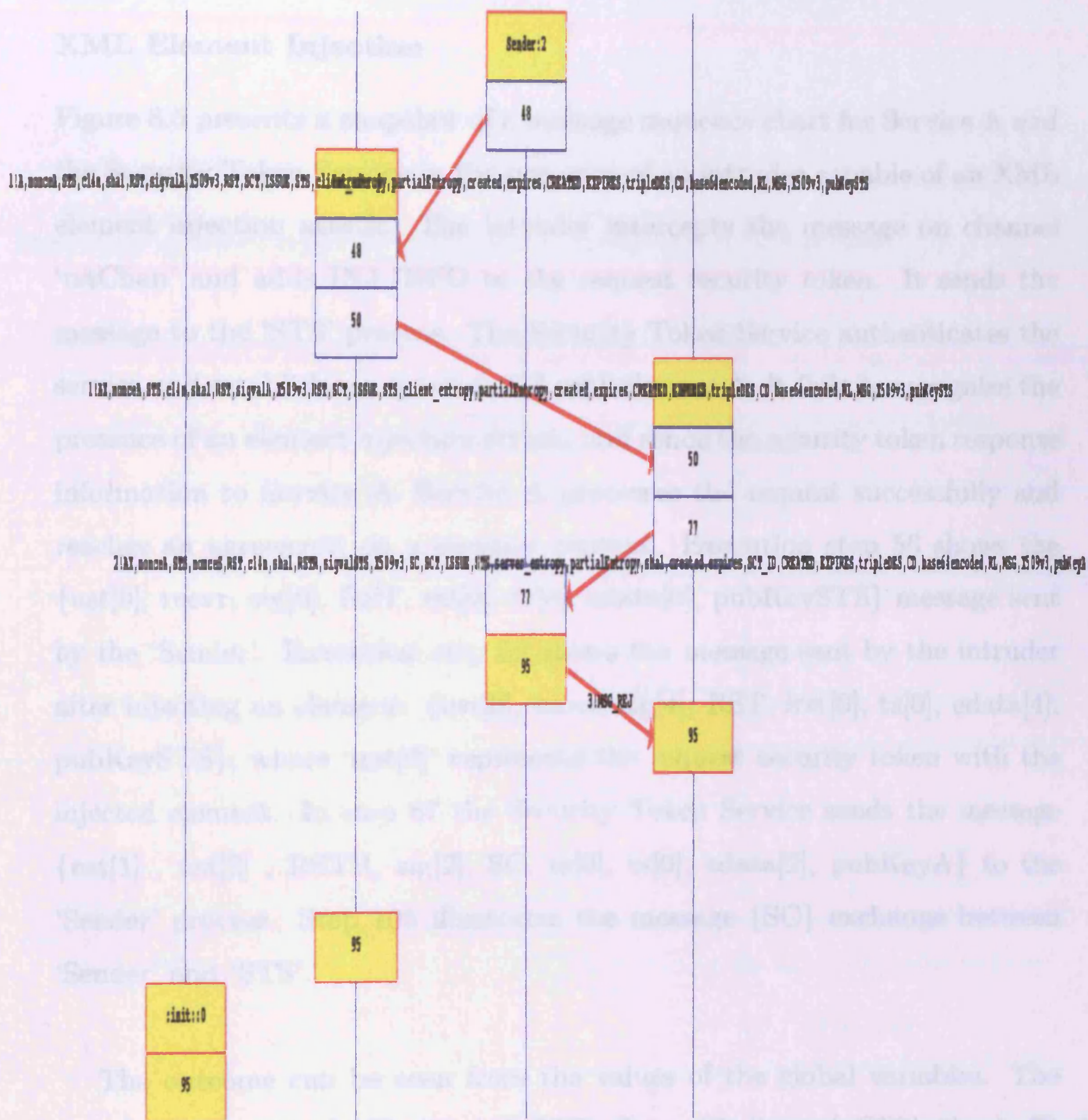


Figure 8.3: Message Sequence Chart for an STP Username Token Injection Attack.

XML Element Injection

Figure 8.5 presents a snapshot of a message sequence chart for Service A and the Security Token Service in the presence of an intruder capable of an XML element injection attack. The intruder intercepts the message on channel 'rstChan' and adds INJ_INFO to the request security token. It sends the message to the 'STS' process. The Security Token Service authenticates the service and establishes a 'partial_SC' with Service A. It fails to recognise the presence of an element injection attack, and sends the security token response information to Service A. Service A processes the request successfully and reaches an agreement on a security context. Execution step 56 shows the {ust[0], recvr, sig[0], RST, rst[0], ts[0], edata[0], pubKeySTS} message sent by the 'Sender'. Execution step 59 shows the message sent by the intruder after injecting an element: {ust[0], recvr, sig[4], RST, irst[0], ts[0], edata[4], pubKeySTS}, where 'irst[0]' represents the request security token with the injected element. In step 87 the Security Token Service sends the message {ust[1] , ust[2] , RSTR, sig[2], SC, rs[0], ts[0], edata[2], pubKeyA} to the 'Sender' process. Step 105 illustrates the message {SC} exchange between 'Sender' and 'STS'.

The outcome can be seen from the values of the global variables. The global variables 'SenderChallengeA_STS', 'RecvrChallengeA_STS', 'SenderBindA_STS' and 'RecvrBindA_STS' are updated to 1, as both the services are able to validate the username token of each other. The Security Token Service is able to agree on a partial security context agreement and updates 'partial_SC' to 1. The information required to reach this agreement is not altered. The

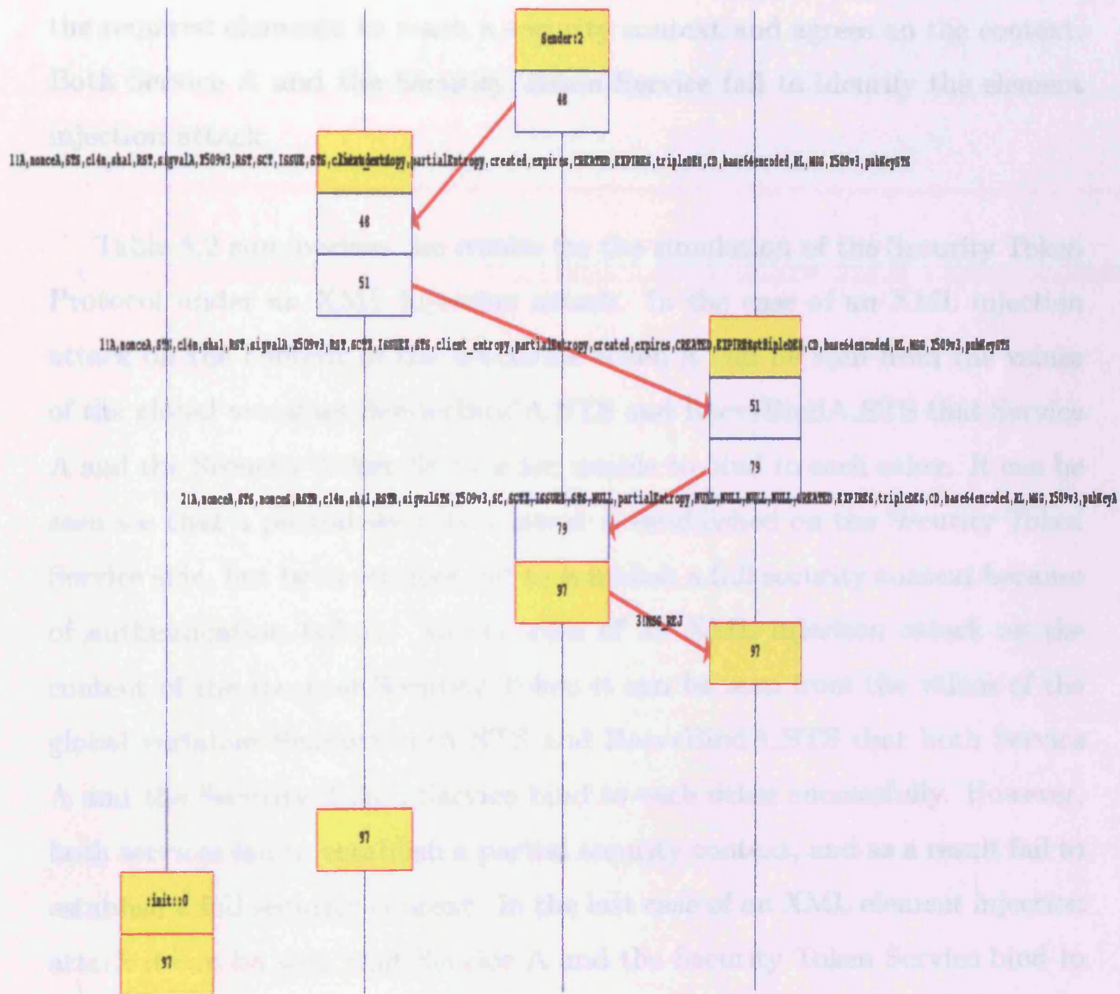


Figure 8.4: Message Sequence Chart for an STP RST Content Injection Attack.

Security Token Service fails to identity the injected element in the request. The Security Token Service sends the security context back to the requestor, Service A, containing all the fields in the original request. Service A processes the required elements to reach a security context and agrees on the context. Both Service A and the Security Token Service fail to identify the element injection attack.

Table 8.2 summarises the results for the simulation of the Security Token Protocol under an XML injection attack. In the case of an XML injection attack on the content of the username token it can be seen from the values of the global variables SenderBindA_STS and RecvrBindA_STS that Service A and the Security Token Service are unable to bind to each other. It can be seen see that a partial security context is established on the Security Token Service side, but both services fail to establish a full security context because of authentication failure. In the case of an XML injection attack on the content of the Request Security Token it can be seen from the values of the global variables SenderBindA_STS and RecvrBindA_STS that both Service A and the Security Token Service bind to each other successfully. However, both services fail to establish a partial security context, and as a result fail to establish a full security context. In the last case of an XML element injection attack it can be seen that Service A and the Security Token Service bind to each other successfully and are able to establish a full security context. The model fails to identify the attack.

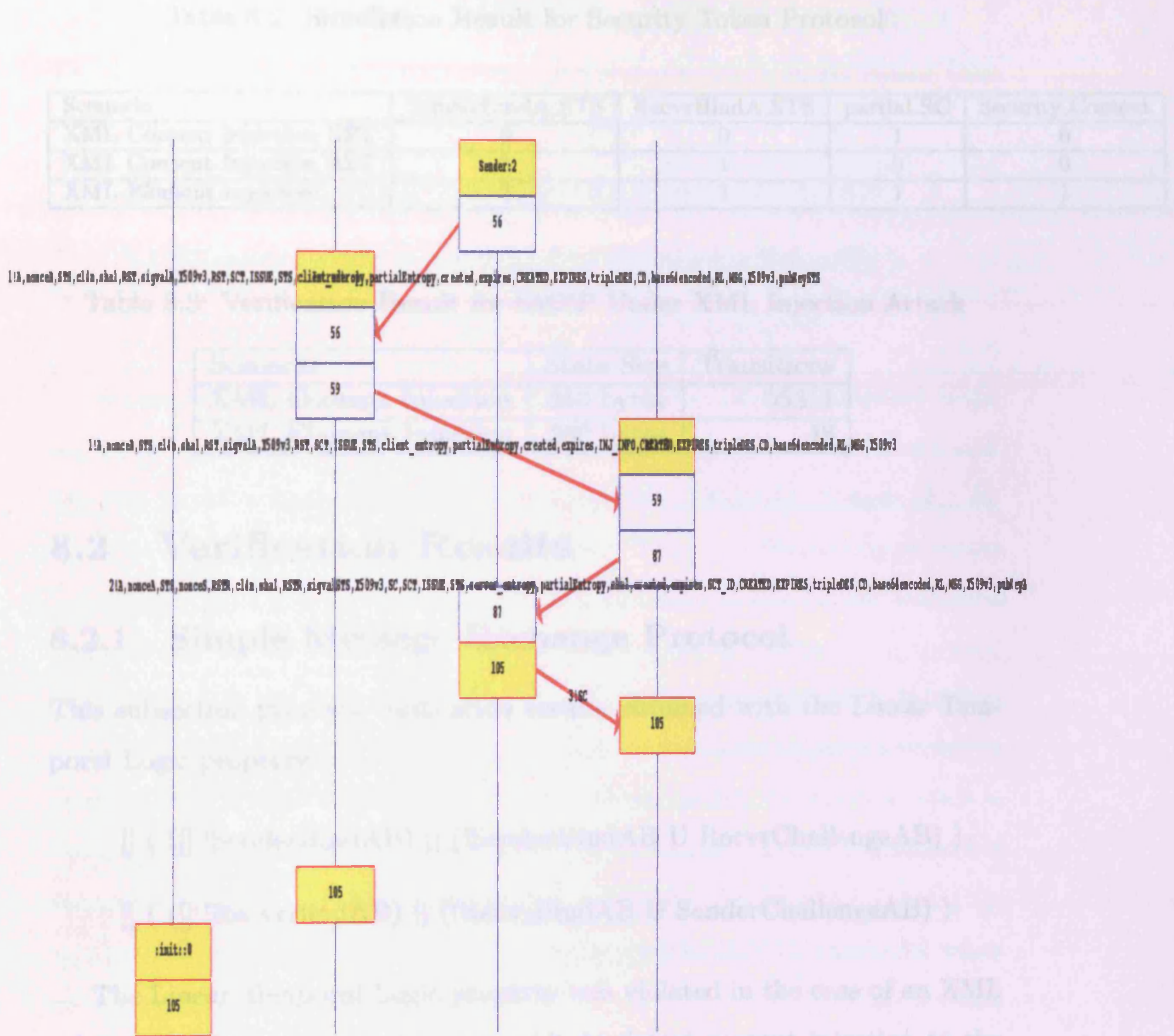


Figure 8.5: Message Sequence Chart for an STP RST Element Injection Attack.

Table 8.2: Simulation Result for Security Token Protocol

Scenario	SenderBindA_STS	RecvrBindA_STS	partial_SC	Security_Context
XML Content Injection UST	0	0	1	0
XML Content Injection RST	1	1	0	0
XML Element Injection	1	1	1	1

Table 8.3: Verification Result for SMEP Under XML Injection Attack

Scenario	State Size	Transitions
XML Content Injection	260 bytes	55311
XML Element Injection	260 bytes	48

8.2 Verification Results

8.2.1 Simple Message Exchange Protocol

This subsection presents verification results obtained with the Linear Temporal Logic property:

[] (([] !SenderBindAB) || (!SenderBindAB U RecvrChallengeAB))

[] (([] !RecvrBindAB) || (!RecvrBindAB U SenderChallengeAB))

The Linear Temporal Logic property was violated in the case of an XML content injection attack. *Spin* was able to detect content injection to the message and generated an error. In the case of element injection to the message, the property was satisfied, showing that *Spin* was unable to identify the illegal element injected into the message. The rest of the results, the depth reached and the memory used, are summarized in Table 8.3.

8.2.2 Security Token Protocol

The following Linear Temporal Logic property for the Security Token Protocol was fed into the *Spin* verification engine:

[] (([] !senderbindAS)|| (!senderbindAS U recvrchallengeAS))

[] (([] !recvrbindAS) || (!recvrbindAS U senderchallengeAS))

[] (([] !securitycontext) || (!securitycontext U partialSC))

It can be seen that there is a violation of the Linear Temporal Logic property in case of content injection. The authentication variables are zero and the security context fails to be established. Similarly, in case of a request security token attack, where the TokenType and RequestType values are changed, it can be seen that there is a violation of the Linear Temporal Logic property. Service A and Service B are able to authenticate to each other but fail to agree on a security context. In the case of an XML Injection attack on the Security Token Protocol it can be seen that there is a violation of the Linear Temporal Logic property, and an invalid verification result is obtained. *Spin* is unable to detect this type of attack. Table 8.4 summarises the memory used and the depth reached results for the Security Token Protocol under an XML injection attack. The results can be employed when analysing the application of various model checkers to Web services based cryptographic protocol.

8.3 Concluding Remarks

Simulations have been conducted on pushdown automata models of the Simple Message Exchange Protocol under an XML injection attack. The Simple

Table 8.4: Verification Result for STP Under XML Injection Attack

Scenario	State Size	Transistions
XML UST Content Injection	328 bytes	65309
XML RST Content Injection	328 bytes	31604
XML Element Injection	336 bytes	297

Message Exchange Protocol is modelled as a three-step protocol as described in Chapter 4. Any deviation from the steps of the protocol results in a change to the expected behaviour of the protocol. Such changes can be seen from the values of the global variables. It has been shown that modelling a protocol in terms of a pushdown automaton not only captures the behaviour of the protocol and its participants, but it also allows the detection of any changes to the correct working of the protocol.

An analysis of the pushdown automaton model of the Security Token Protocol as a three-step protocol has been carried out, as explained in Chapter 4. It was shown that if the Security Token Protocol executed all the three steps for the protocol run successfully, the requesting service was able to bind to the Security Token Service. The requesting service was also able to establish a full security context. Any deviation from the defined behaviour of the Security Token Protocol resulted in failure to achieve binding and/or establishment of full security context.

Pushdown automata allow us to capture the behaviour of the protocols, and for additional analysis and verification of the behaviour Linear Temporal Logic properties were used. First, simulations of the pushdown automata model without the Linear Temporal Logic properties were fed into *Spin*. It

was shown that if a correct protocol run was successfully executed, i.e., all the steps of the run were completed, it was possible to detect any deviations from the correct behaviour from the global variables. This shows the push-down automaton was able correctly to express the behaviour of the Simple Message Exchange Protocol and the Security Token Protocol and can be used for conducting a behavioural analysis of any protocol.

Simulation on the XML injection attack model was performed using *Spin*, and the results were summarised in Tables 8.1 and 8.2. Message sequence charts for the protocol runs were presented, and it was shown that the model is able to detect an XML content injection attack, but fails to identify an XML element injection attack. It was found that *Spin* was able to detect any content injection attacks on the Simple Message Exchange Protocol and the Security Token Protocol, but it was unable to detect element injection attacks on either protocol. For modelling Web services based cryptographic protocols and related attacks we suggest that future model checkers should be designed for Web services, and we plan to extend *Spin* for modelling such protocols in future work.

CHAPTER 9

Conclusions and Future Work

This thesis has studied the design and analysis of Web Services Based Cryptographic Protocols (WSBCPs) as pushdown automaton (PDA) systems, with the aim of using PDAs to model the correct operation of WSBCPs, and to effectively reflect the properties of these protocols. Automaton based models for WSBCPs have been developed, and we suggest that modelling such protocols using automata may be more suitable for Web services based models, as they allow more detailed tracking of the protocol behaviour and the detection of alterations to this behaviour. PDAs allow the properties of the WSBCPs to be mapped, which allows the verification of the properties of the system without specifying them in another language, such as Linear Temporal Logic or pi-calculus. An intruder model for the Simple Message Exchange Protocol and the Security Token Protocol has been developed based on the classic Dolev-Yao model. Both these protocols were subjected to attacks by

an intruder. An extension to the Dolev-Yao model has been given encompassing XML injection attacks. We have listed the results of this model, and verified our model using *Spin*, a general-purpose model checker.

The approach presented in this thesis allows feasible modelling techniques for theoretically and practically modelling Web services based protocols. This conclusion is supported by the following activities and outcomes presented in this thesis:

- The development of two novel WSBCP's based on the WS-Security and WS-Trust specifications. These protocols are based on rules suggested for modelling cryptographic protocols. The goals for these protocols have been defined and their properties modelled as Linear Temporal Logic formulas.
- The modelling of the environment for these protocols as transition systems. The environment depicts all the possible actions that can be taken by the principals. The environment is susceptible to attacks based on a Dolev-Yao intruder.
- The modelling of the WS-Security and WS-Trust based protocols as two-stack pushdown automata. Automaton models for WSBCP's describe the behaviour in detail. Two-stack pushdown automata are used to map the correct operations of the protocols.
- We argue for the suitability of applying a general-purpose model checker to WSBCPs. The automaton models are translated into Promela, the input language to *Spin*. To the best of our knowledge the Promela models presented in this thesis are the first for WSBCPs.

- Both the protocols have been verified for authentication and secrecy in the presence of an intruder. For the WS-Trust based protocol, the establishment of a security context has been verified. The intruder is allowed to learn certain information and on the basis of it acts as an impostor, or performs blocking attacks on these protocols.
- We have extended the capabilities of the Dolev-Yao intruder model. The intruder is not only able to perform the basic Dolev-Yao operations, but also is able to carry out basic XML injection attacks on the Simple Message Exchange Protocol and the Security Token Protocol.
- XML injection attacks can be classified as *(i)* content alteration attacks, and *(ii)* element alteration attacks. The intruder is allowed to alter the content of some parts of the message. He is also allowed to add an element to the message. The Simple Message Exchange Protocol and the Security Token Protocol have been subjected to this attack model.

Concluding Remarks

The behavious of WSBCPs can be modelled using multi-stack pushdown automata. The PDA model not only reflects the working of the protocols, but also the properties these protocols are supposed to possess. The modelling technique can be used to specify the properties of the protocol under study without the need for a property specification language. The PDA model for WSBCPs also allows us to detect any deviation from the expected behaviour of the protocols.

We modelled the SMEP and STP protocols in terms of Promela. To the best of our knowledge this is the first attempt to model WSBCPs in

Promela. We produced an intruder model for the Simple Message Exchange Protocol and the Security Token Protocol based on the Dolev-Yao model, specifically targeted for WSBCPs. The Simple Message Exchange Protocol and the Security Token Protocol were subjected to Dolev-Yao intruder attacks and verified.

The Dolev-Yao model should be extended for WSBCPs and formalised. The model should be extended to reflect the attacks specific to Web services, i.e.,

1. Attacks on confidentiality. The objective of a confidentiality attack is to force the targeted application to disclose information that the attacker is not authorised to see, including sensitive information and private information. The XML Encryption, WS-Security, and HTTPS standards provide confidentiality protection for Web services. WS-Security and HTTPS are generally used to protect the confidentiality of SOAP messages in transit, leaving data at rest vulnerable to attack.
2. Attacks on integrity. The objective of an integrity attack is to exploit the targeted application to make unauthorised changes to information accessed/handled by the application. Web service standards for protecting the integrity of data include WS-Security and XML Signature.
3. Command injection. In a command injection executable logic is inserted into non-executable text strings submitted to a Web service. The main types of command injection are SQL injection targeting Web service-enabled database applications, and XML injection targeting Web services.
4. Reconnaissance attacks. Reconnaissance attacks have the objective of

collecting information about an application and its execution environment to better target other types of attacks at that application. There are no standards for preventing reconnaissance attacks, e.g., dictionary attacks, WSDL scanning, sniffing, etc.

5. Privilege escalation attack. The objective of privilege escalation attacks is to enable the attacker to change the privilege level of a process, thereby taking control of that now-compromised process to bypass security controls that would otherwise limit the attacker's access to the Web service's functionality, data, resources, and environment. For example, format string attacks and exploiting unprotected administrative interfaces.

6. Denial-of-Service, malicious code attacks, etc. With the increasing capabilities of the intruder, unbroken encryption can no longer be assumed.

We have presented an extension to the Dolev-Yao model targeted specifically for WSBCPs by adding the capability to carry out XML injection attacks. To the best of our knowledge this is the first formal extension of the Dolev-Yao model for XML injection attacks.

We conclude that, although general-purpose model checkers can analyse and verify the general working of WSBCPs, for detailed analysis and verification of the behaviour of WSBCPs work needs to be done on model checkers specific to Web services based security protocols. We have presented a Promela model for the Simple Message Exchange Protocol and the Security Token Protocol, along with an intruder model. We have performed simulations and verification of our model using *Spin* and presented the results in

Chapter 6.

Limitations of our Model

This subsection reviews the complications in modelling the pushdown automaton based protocol models (the Simple Message Exchange Protocol and the Security Token Protocol) in the Promela language. Our approach deviates from traditional approaches to modelling automaton using languages. Instead, we model the Simple Message Exchange Protocol and the Security Token Protocol as steps. The set of steps in the protocol run is represented by an 'atomic' structure, embedded with functions applied in that step. We faced difficulty in modelling the input tape environment. The input tape is presented as a combination of symbols read from the messages on the channel and hard-coded input reads. We divide some functionality of the services into smaller parts, so they can be used when the model is extended to incorporate XML injection attacks. We were also unable to map the dynamic nature of the XML envelope. The XML structures could only be modelled as static 'typedef' structures.

We extended the Dolev-Yao model to carry out XML injection attacks. We saw that the Promela model did not allow us to detect element injection attacks in some cases. Our future work will allow us to address this issue. To counter the current limitations of our work, we propose to work on the extension of both the Dolev-Yao model and the *Spin* model checker.

Future Work

The Dolev-Yao model has been adopted for analysis of a large number of cryptographic protocols. With the emergence of new types of attacks and the nature of the Web services based cryptographic protocols, the model should be expanded. There have been some proposed suggestions but no formal work has been proposed to the best of our knowledge. We propose to continue our work on an extension of the Dolev-Yao model for WSBCPs encompassing various attacks (Command Injection, Attacks on Confidentiality, and Attacks on Integrity) targeted for WSBCPs. We will work towards developing a formalism for this.

As a next step we also propose to work on a Web services protocol model checker. The idea is to allow WSBCPs to be modelled using pushdown automata that reflect their operation and properties. We give a brief overview of the framework for the model in Figure 9.1. All implementation modules are C-based codes. The XML parser module, as its name suggests, parses a given SOAP document containing the XML signatures. The purpose of this parser is to automatically extract relevant information from the SOAP traffic and obtain the corresponding WS-* security tags for further processing. The input of this module is the SOAP messages carrying the WS-* traffic, and its output is the populated data structures which are required in subsequent stages for making the automaton. This module extracts the rule set from the given security specification which we want to analyse. This module is semi-automatic as some manual rule extraction is required to get the exhaustive lists. The purpose of this module is to generate the security requirements. The requirements are presented in files in a format which our automaton modules can understand. The PDA Module is the core module

in which we generate the proposed multi-stack pushdown automaton for a given Web service traffic scenario. The outputs of the XML Parser module and the Design module are used as inputs in this stage. The output of this stage is an automated pushdown automaton which represents the Web service traffic along with any loopholes which may be part of the security specification. The PDA module's output is then subjected to the validation module which verifies the security requirements. The validation engine will convert the model into Promela and will be fed into Spin for verification. This module will also point out any security vulnerabilities present in the specification. The Reporting module is a standard reporting module which populates necessary logs and suggests possible courses of action where required. The results from the model checker will be fed into *Spin* to verify the results.

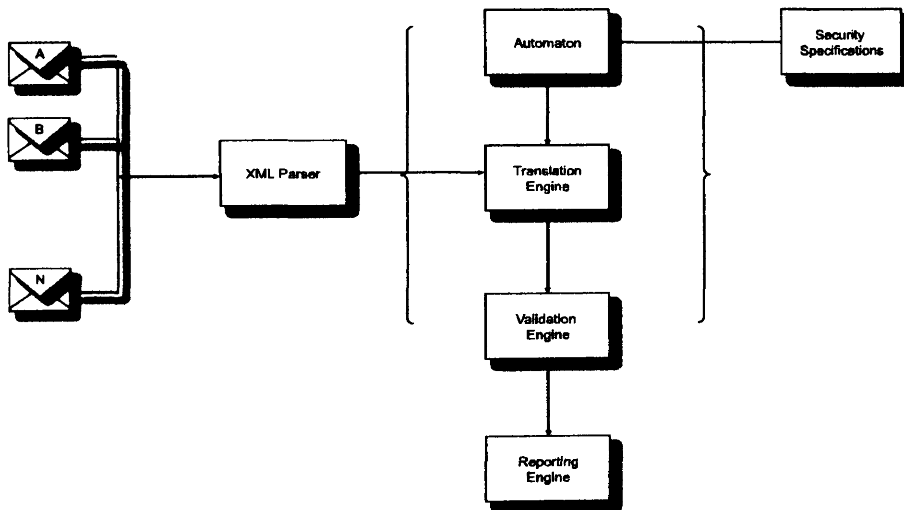


Figure 9.1: Proposed architecture

Bibliography

[Aba99] M. Abadi. Secrecy by typing in security protocols. *J. ACM*, 46(5):749–786, 1999.

[ABea05] A. Armando, D. Basin, and et al. The AVISPA tool for the automated validation of internet security protocols and applications. In *Proceedings of CAV'2005*, LNCS 3576, pages 281–285. Springer-Verlag, 2005.

[ACN11] C. Andrés, M. E. Cambronero, and M. Núñez. Passive Testing of Web Services. *Lecture Notes in Computer Science*, 6551:56–+, 2011.

[Aea05] S. Anderson and et al. Web services secure conversation language (ws-secureconversation). Technical report, OASIS, February 2005.

[AL00] R M. Amadio and D. Lugiez. On the reachability problem in cryptographic protocols. In *Proceedings of the 11th Inter-*

national Conference on Concurrency Theory, CONCUR '00, pages 380–394, London, UK, 2000. Springer-Verlag.

[AVA] AVANTSSAR. Automated validation of trust and security-oreinted architectures.

[BAN90] M. Burrows, M. Abadi, and R. Needham. A logic of authentication. *ACM Trans. Comput. Syst.*, 8:18–36, February 1990.

[BCFG04] K. Bhargavan, R. Corin, C. Fournet, and A. Gordon. Secure sessions for web services. In *SWS '04: Proceedings of the 2004 workshop on Secure web service*, pages 56–66, New York, NY, USA, 2004. ACM Press.

[BCJS04] F. Butler, I. Cervesato, A. Jaggard, and A. Scedrov. A formal analysis of some properties of kerberos 5 using msr, 2004.

[Bea04] G. Behrmann and et al. A tutorial on uppaal. In *Fourth International School on Formal Methods for the Design of Computer, Communication, and Software Systems*, pages 200–236. Springer, 2004.

[Bea05] R. Bilorusets and et al. Web services reliable messaging protocol (ws-reliablemessaging). Technical report, BEA Systems, IBM, Microsoft Corporation, Inc, and TIBCO Software Inc.., February 2005.

[Bea06a] M. Backes and et al. Symbolic and cryptographic analysis of the secure ws-reliablemessaging scenario. In *In Foundations of Software Science and Computation Structures*, pages 428–445. Springer-Verlag, 2006.

[Bea06b] S. Bajaj and et al. Web services policy 1.2 - framework (ws-policy). Technical report, W3C, April 2006.

[BF04] K. Bhargavan and C. Fournet. Tulafale: A security tool for web services, 2004.

[BFG06] K. Bhargavan, C. Fournet, and A. Gordon. Verified reference implementations of ws-security protocols. In *In 3rd International Workshop on Web Services and Formal Methods (WS-FM 2006), volume 4184 of LNCS*, pages 88–106. Springer, 2006.

[BFGT06] K. Bhargavan, C. Fournet, A. Gordon, and S. Tse. Verified interoperable implementations of security protocols. In *CSFW '06: Proceedings of the 19th IEEE workshop on Computer Security Foundations*, pages 139–152, Washington, DC, USA, 2006. IEEE Computer Society.

[BG05] M. Backes and T. Groβ. Tailoring the dolev-yao abstraction to web services realities. In *SWS '05: Proceedings of the 2005 workshop on Secure web services*, pages 65–74, New York, NY, USA, 2005. ACM.

[BK08] C. Baier and J-P. Katoen. *Principles of Model Checking (Representation and Mind Series).* The MIT Press, 2008.

[Bla01] B. Blanchet. An efficient cryptographic protocol verifier based on prolog rules. *Computer Security Foundations Workshop, 2001. Proceedings. 14th IEEE*, pages 82–96, 2001.

[Bla02] B. Blanchet. From secrecy to authenticity in security protocols. In *In 9th International Static Analysis Symposium (SAS02*, pages 342–359. Springer, 2002.

[But01] R. W. Butler. What is formal methods?, 2001.

[CC11] D. Christiansen and M. Carbone. Formal semantics and implementation of bpmn 2.0 inclusive gateways. In *Proceedings of the 7th international conference on Web services and formal methods*, WS-FM'10, pages 146–160, Berlin, Heidelberg, 2011. Springer-Verlag.

[CDL+99] I. Cervesato, N. A. Durgin, P. D. Lincoln, J. C. Mitchell, and A. Scedrov. A meta-notation for protocol analysis. In *Proceedings of the 12th IEEE workshop on Computer Security Foundations*, CSFW '99, pages 55–69, Washington, DC, USA, 1999. IEEE Computer Society.

[Cea04] E. Christensen and et al. Web services addressing (ws-addressing). Technical report, BEA Systems, IBM, Microsoft Corporation, Inc, and TIBCO Software Inc.., August 2004.

[Cea07] Y. Chevalier and et al. Towards an automatic analysis of web service security. In *Proceedings of the 6th international symposium on Frontiers of Combining Systems*, FroCoS '07, pages 133–147, Berlin, Heidelberg, 2007. Springer-Verlag.

[Coh00] E. Cohen. Taps: a first-order verifier for cryptographic protocols. In *Computer Security Foundations Workshop, 2000. CSFW-13. Proceedings. 13th IEEE*, pages 144 –158, 2000.

[Con] The Web Application Security Consortium. Xml injection attack.

[DK99] Z. Dang and R A. Kemmerer. Using the astral model checker to analyze mobile ip. In *Proceedings of the 21st international*

conference on Software engineering, ICSE '99, pages 132–141, New York, NY, USA, 1999. ACM.

[DLea05] G. Della-Libera and et al. Web services security policy language (WS-SecurityPolicy), July 2005.

[DM00] G. Denker and J. Millen. Capsl integrated protocol environment. In *In Proc. of DARPA Information Survivability Conference (DISCEX 2000), pp 207-221, IEEE Computer Society*, pages 207–221. IEEE Computer Society, 2000.

[DPC+06] G. Diaz, J-J Pardo, M. Cambronero, V. Valero, and F. Cuartero. Verification of web services with timed automata. *Electronic Notes in Theoretical Computer Science*, 157(2):19–34, 2006.

[DS97] B. Dutertre and S. Schneider. Using a pvs embedding of csp to verify authentication protocols. In Elsa Gunter and Amy Felty, editors, *Theorem Proving in Higher Order Logics*, volume 1275 of *Lecture Notes in Computer Science*, pages 121–136. Springer Berlin / Heidelberg, 1997. 10.1007/BFb0028390.

[DY81] D. Dolev and A. C. Yao. On the security of public key protocols. In *Proceedings of the 22nd Annual Symposium on Foundations of Computer Science*, pages 350–357, Washington, DC, USA, 1981. IEEE Computer Society.

[EG83] S. Even and O. Goldreich. On the security of multi-party ping-pong protocols. In *Proceedings of the 24th Annual Symposium on Foundations of Computer Science*, pages 34–39, Washington, DC, USA, 1983. IEEE Computer Society.

[ERS01] D. Eastlake, 3rd, J. Reagle, and D. Solo. Xml-signature syntax and processing, 2001.

[FA01] M. Fiore and M. Abadi. Computing symbolic models for verifying cryptographic protocols. In *In Proc. of the 14th Computer Security Foundation Workshop (CSFW14*, pages 160–173. IEEE, Computer Society Press, 2001.

[Fea04] X. Fu and et al. Wsat: A tool for formal analysis of web services. In *the Proc. of 16th Int. Conf. on Computer Aided Verification (CAV*, pages 510–514. Springer, 2004.

[FHG98] F.J.T. Fabrega, J.C. Herzog, and J.D. Guttman. Strand spaces: why is a security protocol correct? In *Security and Privacy, 1998. Proceedings. 1998 IEEE Symposium on*, pages 160 –171, May 1998.

[For94] W. Ford. *Computer communications security: principles, standard protocols and techniques.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1994.

[Fu11] Xiang Fu. Conformance verification of privacy policies. In *Proceedings of the 7th international conference on Web services and formal methods*, WS-FM'10, pages 86–100, Berlin, Heidelberg, 2011. Springer-Verlag.

[Ger] R. Gerth. Concise promela reference.

[GJ03] A. Gordon and A. Jeffrey. Authenticity by typing for security protocols. *J. Comput. Secur.*, 11(4):451–519, 2003.

[GP02] A. Gordon and R. Pucella. Validating a web service security abstraction by typing. In *XMLSEC '02: Proceedings of the 2002 ACM workshop on XML security*, pages 18–29, New York, NY, USA, 2002. ACM.

[HC98] D. Harkins and D. Carrel. The internet key exchange (ike), 1998.

[Hea02] T. Henzinger and et al. Lazy abstraction. In *In POPL*, pages 58–70. ACM Press, 2002.

[HG01] M.L Hui and Gavin. Fault-preserving simplifying transformations for security protocols. *Journal of Computer Security*, 9(1/2):3–46, 2001.

[HM06] H. Huang and R. Mason. Model checking technologies for web services. *Software Technologies for Future Embedded and Ubiquitous Systems, and International Workshop on Collaborative Computing, Integration, and Assurance, The IEEE Workshop on*, 0:217–224, 2006.

[HM11] K M. Hee and A J. Mooij. Soundness-preserving refinements of service compositions. In *Proceedings of the 7th international conference on Web services and formal methods*, WS-FM'10, pages 131–145, Berlin, Heidelberg, 2011. Springer-Verlag.

[HMU06] J E. Hopcroft, R Motwani, and J D. Ullman. *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.

[Hol03] G. Holzmann. *The Spin model checker: primer and reference manual.* Addison-Wesley Professional, 2003.

[HS00] J. Heather and S. Schneider. Towards automatic verification of authentication protocols on an unbounded network, 2000.

[HT96] N. Heintze and J. D Tygar. A model for secure protocols and their compositions. *IEEE Transactions on Software Engineering*, 22:2–13, 1996.

[Hui99] A. Huima. Efficient infinite-state analysis of security protocols. In *Proc. FLOC'99 Workshop on Formal Methods and Security Protocols*, 1999.

[JJLea04] J. Johnson, J. Johnson, D. Langworthy, and et al. Formal specification of a web services protocol. In *University of Pisa*, pages 147–158. Elsevier, 2004.

[Jou09] A. Joux. *Algorithmic Cryptanalysis.* Chapman & Hall/CRC, 2009.

[KM08] F. Krger and S. Merz. *Temporal Logic and State Systems (Texts in Theoretical Computer Science. An EATCS Series).* Springer Publishing Company, Incorporated, 1 edition, 2008.

[KR05] E. Kleiner and A.W . Roscoe. On the relationship between web services security and traditional protocols, 2005.

[KS11] Esra Kucukoguz and Jianwen Su. On lifecycle constraints of artifact-centric workflows. In *Proceedings of the 7th international conference on Web services and formal methods*, WS-FM'10, pages 71–85, Berlin, Heidelberg, 2011. Springer-Verlag.

[Lea06] H. Lockhart and et al. WS-Federation 1.1. Technical report, OASIS, December 2006.

[Lin06] P. Linz. *An introduction to formal languages and automata.* Jones and Bartlett Publishers Inc., London, U.K., 2006.

[Low96a] G. Lowe. Breaking and fixing the needham-schroeder public-key protocol using fdr. In Tiziana Margaria and Bernhard Steffen, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 1055 of *Lecture Notes in Computer Science*, pages 147–166. Springer Berlin / Heidelberg, 1996.

[Low96b] G. Lowe. Breaking and fixing the needham-schroeder public-key protocol using fdr. In *In Tools and Algorithms for the Construction and Analysis of Systems*, pages 147–166. Springer-Verlag, 1996.

[Low97] G. Lowe. A hierarchy of authentication specifications. *IEEE Computer Security Foundations Workshop*, 0:31, 1997.

[LW09] M. Little and A. Wilkinson. Web services atomic transaction (WS-AtomicTransaction) version 1.2. Technical report, OASIS, February 2009.

[MCF87] J.K. Millen, S.C. Clark, and S.B. Freedman. The interrogator: Protocol secuity analysis. *Software Engineering, IEEE Transactions on*, SE-13(2):274 - 288, feb. 1987.

[MCJ97] W. Marrero, E. Clarke, and S. Jha. A model checker for authentication protocols. In *Rutgers University*, 1997.

[Mea92] C. Meadows. Applying formal methods to the analysis of a key management protocol. *Journal of Computer Security*, 1, 1992.

[Mea00] C. Meadows. Open issues in formal methods for cryptographic protocol analysis. In *In Proceedings of DISCEX 2000*, pages 237–250. IEEE Computer Society Press, 2000.

[Mea03] C. Meadows. Formal methods for cryptographic protocol analysis: emerging issues and trends. *Selected Areas in Communications, IEEE Journal on*, 21(1):44–54, Jan 2003.

[MMS97] J.C. Mitchell, M. Mitchell, and U. Stern. Automated analysis of cryptographic protocols using mur phi;. In *Security and Privacy, 1997. Proceedings., 1997 IEEE Symposium on*, pages 141–151, May 1997.

[MR00] J. Millen and H. Rue. Protocol-independent secrecy. In *In 2000 IEEE Symposium on Security and Privacy. IEEE Computer Society*, pages 110–119. Society Press, 2000.

[MS01] J. Millen and V. Shmatikov. Constraint solving for bounded-process cryptographic protocol analysis, 2001.

[NB11] D. Nadkarni and S. Basu. Failure analysis for composition of web services represented as labeled transition systems. In *Proceedings of the 7th international conference on Web services and formal methods*, WS-FM'10, pages 161–175, Berlin, Heidelberg, 2011. Springer-Verlag.

[Nea05] H. Nezhad and et al. Securing service-based interactions: Issues and directions. *IEEE Distributed Systems Online*, 2005.

[Nei03] M. Neill. *Web Services Security*. McGraw-Hill/Osborne, 2003.

[NGG+07] A. Nadalin, M. Goodner, M. Gudgin, A. Barbir, and H. Granqvist. WS-Trust 1.3. Technical report, OASIS, March 2007.

[NKHBM04] A. Nadalin, C. Kaler, P. Hallam-Baker, and R. Monzillo. Web services security: SOAP message security 1.0 (WS-Security 2004). Technical report, OASIS, March 2004.

[NS78] Roger M. Needham and Michael D. Schroeder. Using encryption for authentication in large networks of computers. *Commun. ACM*, 21:993–999, December 1978.

[Pau97] L. Paulson. Proving properties of security protocols by induction. In *In 10th IEEE Computer Security Foundations Workshop*, pages 70–83. IEEE Computer Society Press, 1997.

[Pau98] L C. Paulson. The inductive approach to verifying cryptographic protocols. *J. Comput. Secur.*, 6:85–128, January 1998.

[Pau99] L. Paulson. Inductive analysis of the internet protocol tls (transcript of discussion). In *Proceedings of the 6th International Workshop on Security Protocols*, pages 13–23, London, UK, 1999. Springer-Verlag.

[pol]

[pot]

[Rea02] J. Reagle. XML encryption requirements. Technical report, W3C, March 2002.

[Res] Microsoft Research. Samoa: Formal tools for securing web services.

[RR04] J. Rosenberg and D. Remy. *Securing Web Services with WS-Security: Demystifying WS-Security, WS-Policy, SAML, XML Signature, and XML Encryption.* Pearson Higher Education, 2004.

[RRM] M. Rennhard, S. Rafaeli, and L. Mathy. From set to pset - the pseudonymous secure electronic transaction protocol.

[SBP01] D Song, S Berezin, and A Perrig. Athena: a novel approach to efficient automatic security protocol analysis. *Journal of Computer Security*, 9:2001, 2001.

[Sch95] B. Schneier. *Applied cryptography (2nd ed.): protocols, algorithms, and source code in C.* John Wiley & Sons, Inc., New York, NY, USA, 1995.

[Sch98] S. Schneider. Verifying authentication protocols in csp. *Software Engineering, IEEE Transactions on*, 24(9):741–758, Sep 1998.

[Sta95] W. Stallings. *Network and internetwork security: principles and practice.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1995.

[Sti02] D. Stinson. *Cryptography: Theory and Practice,Second Edition.* CRC/C&H, 2nd edition, 2002.

[TCCD07] L. Tobarra, D. Cazorla, F. Cuartero, and G. Diaz. Analysis of web services secure conversation with formal methods. *Internet*

and Web Applications and Services, 2007. ICIW '07. Second International Conference on, pages 27–27, May 2007.

[TCM02] I. Takeshi, A. Clark, and H. Maruyama. A stream-based implementation of xml encryption. In *Proceedings of the 2002 ACM workshop on XML security*, XMLSEC '02, pages 11–17, New York, NY, USA, 2002. ACM.

[TP05] W. Tsai and R. Paul. Proof slicing with application to model checking web services. In *Proceedings of the Eighth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, ISORC '05, pages 292–299, Washington, DC, USA, 2005. IEEE Computer Society.

[WEW11] Matthias Weidlich, Felix Elliger, and Mathias Weske. Generalised computation of behavioural profiles based on petri-net unfoldings. In *Proceedings of the 7th international conference on Web services and formal methods*, WS-FM'10, pages 101–115, Berlin, Heidelberg, 2011. Springer-Verlag.

[WL93] T. Woo and S. Lam. A semantic model for authentication protocols, 1993.

Appendix: SMEP

This appendix presents the macros and global variables used in the Promela model of the Simple Message Exchange Protocol.

SMEP Macros

```
#define senderchallenge(x,y) if
:: (x==A && y==B) → senderchallengeAB =1
:: (x==A && y==I) → senderchallengeAI=1
:: (x==I && y==B) → senderchallengeIB=1
:: else skip
fi
#define senderbind(x,y) if
:: ((x==A)&&(y==B))→ senderbindAB=1
:: ((x==A)&&(y==I)) → senderbindAI=1
:: ((x==I)&&(y==B))→ senderbindIB=1
:: else skip
fi
#define recvrchallenge(x,y) if
::((x==A) && (y==B)) → recvrchallengeAB=1
::((x==A) && (y==I)) → recvrchallengeAI=1
::((x==I) && (y==B)) → recvrchallengeIB=1
:: else skip
fi
#define recvrbind(x,y) if
:: ((x==A)&&(y==B))→ recvrbindAB=1
```

```
:: ((x==A)&&(y==I))→ recvrbindAI=1
:: ((x==B)&&(y==I)) → recvrbindIB=1
:: else skip
fi
#define Decryption(pub,priv) if
:: ((pub == pubKeyA)&&(priv == privKeyA))→ valid_DecryptA =1
:: ((pub == pubKeyB)&&(priv == privKeyB))→ valid_DecryptB =1
:: ((pub == pubKeyI)&&(priv == privKeyA))→ valid_DecryptI =1
:: else skip
fi
#define k(x1) if
:: (x1 == nonceA)→ kNa = 1 ;learn_kNa = 1;
:: (x1 == nonceB)→ kNb = 1; learn_kNb = 1;
:: else skip
fi
#define VerifySignature(a,b,c,x,y) if
:: ((a== c14n)&&(b== sha1)&&(c== REQ)&&(x== sigvalA)&&(y==
X509v3)) → valid_dsSigA_REQ=1
:: ((a== c14n)&&(b== sha1)&&(c== ACCEPT)&&(x==
sigvalA)&&(y== X509v3)) → valid_dsSigA_ACK=1
:: ((a== c14n)&&(b== sha1)&&(c== ACK)&&(x== sigvalB)&&(y==
X509v3)) → valid_dsSigB=1
:: ((a== c14n)&&(b== sha1)&&(c== REQ)&&(x== sigvalI)&&(y==
X509v3)) →valid_dsSigI_REQ=1
:: ((a== c14n)&&(b== sha1)&&(c== ACCEPT)&&(x==
sigvalI)&&(y== X509v3)) → valid_dsSigI_ACCEPT=1
:: ((a== c14n)&&(b== sha1)&&(c== ACK)&&(x== sigvalI)&&(y==
X509v3)) → valid_dsSigI_ACCEPT=1
:: else skip
fi
#define isEncrypted(em,cd,cv,rl,dr,ki) if
:: ((em== tripleDES)&&(cd== CD)&&(cv== base64encoded)&&(rl==
RL)&& (dr== MSG)&&(ki== X509v3))→ isEncrypted = isEncrypted + 1
:: else skip
fi
#define VerifyFreshness(cr,ex) if
:: ((cr == CREATED)&&(ex == EXPIRES))→ valid_TimeStamp =1
:: else skip
fi
```

SMEP Global Variables

```
bit senderchallengeAB=0;
bit senderbindAB=0;
bit recvrchallengeAB=0;
bit recvrbindAB=0;
bit senderchallengeAI=0;
bit senderbindAI=0;
bit recvrchallengeAI=0;
bit recvrbindAI=0;
bit senderchallengeIB=0;
bit senderbindIB=0;
bit recvrchallengeIB=0;
bit recvrbindIB=0;
bit valid_DecryptA = 0;
bit valid_DecryptB = 0;
bit valid_DecryptI =0;
int isEncrypted = 0;
bit valid_TimeStamp =0;
bit valid_dsSigA_REQ= 0;
bit valid_dsSigA_ACK= 0;
bit valid_dsSigB = 0;
bit valid_dsSigI_REQ= 0;
bit valid_dsSigI_ACK= 0;
bit valid_dsSigI_ACCEPT= 0;
bit learn_kNa = 0;
bit learn_kNb = 0;
```

Appendix: STP

This appendix presents the macros and global variables used in the Promela model of the Security Token Protocol.

STP Macros

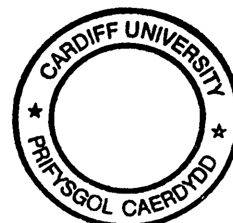
```
#define SenderChallenge(x,y) if
:: (x==A && y==STS) → SenderChallengeA_STS=1
:: (x==I && y==STS) → SenderChallengeI_STS=1
:: else skip
fi
#define AuthenticateRequest(x) if
:: (x==RST) → MsgType_RST=1
:: else skip
fi
#define SenderBind(x,y) if
:: (x==A && y==STS) →SenderBindA_STS=1
:: (x==I && y==STS) → SenderBindI_STS=1
:: else skip
fi
#define RecvrChallenge(x,y) if
:: (x==A && y==STS) → RecvrChallengeA_STS=1
:: (x==I && y==STS) → RecvrChallengeI_STS=1
:: else skip
fi
#define AuthenticateResponse(x) if
```

```
:: (x==RSTR) → MsgType_RSTR=1
:: else skip
fi
#define RecvrBind(x,y) if
::((x==A) && (y==STS)) → RecvrBindA_STS=1
::((x==I) && (y==STS)) →RecvrBindI_STS=1
:: else skip
fi
#define VerifySignature(a,b,c,x,y) if
:: ((a== c14n)&&(b== sha1)&&(c== RST)&&(x== sigvalA)&&(y==
X509v3)) → valid_dsSigA=1
:: ((a== c14n)&&(b== sha1)&&(c== RSTR)&&(x==
sigvalSTS)&&(y== X509v3)) → valid_dsSigSTS=1
:: ((a== c14n)&&(b== sha1)&&(c== RST)&&(x== sigvalI)&&(y==
X509v3)) → valid_dsSigI=1
:: else skip
fi
#define Decryption(pub,priv) if
:: ((pub == pubKeyA)&&(priv == privKeyA))→ valid_DecryptA =1
:: ((pub == pubKeySTS)&&(priv == privKeySTS))→ valid_DecryptSTS
=1
:: ((pub == pubKeyI)&&(priv == privKeyI))→ valid_DecryptI =1
:: else skip
fi
#define VerifyKey(x) if
:: (x == pubKeyA)→ valid_pubKeyA =1
:: (x == pubKeySTS)→ valid_pubKeySTS =1
:: (x == pubKeyA)→ valid_pubKeyI =1
:: else skip
fi
#define PartialSCAgreement(a,b,c,x,y) if
:: ((a== STS)&&(b== SCT)&&(c== ISSUE)&&(x==
partialEntropy)&& (y== client'entropy)) → partial_SC=1
:: else skip
fi
#define SCAgreement(at,tt,rt,em,ce,se,ck,ex,stsn,sts,sdr,scid) if
:: ((at == STS)&&(tt == SCT)&&(rt == ISSUE)&&(em ==
partialEntropy)&& (ce == client'entropy)&&(se == server'entropy)&&(ck
== sha1)&& (ex == expires)&&(stsn == nonceS)&&(sts == STS)&&(sdr
== A)&& (scid == SCT_ID)) → Security_Context=1
:: ((at == STS)&&(tt == SCT)&&(rt == ISSUE)&&(em ==
```

```
partialEntropy)&& (ce == client˙entropy)&&(se == server˙entropy)&&(ck
== sha1)&& (ex == expires)&&(stsn == nonceS)&&(sts == STS)&&(sdr
== I)&& (scid == SCT_ID)) → Security_Context_I=1
::else skip;
fi
#define isEncrypted(em,cd,cv,rl,dr,ki) if
:: ((em== tripleDES)&&(cd== CD)&&(cv== base64encoded)&&(rl==
RL) &&(dr== MSG)&&(ki== X509v3))→ isEncrypted = isEncrypted + 1
:: else skip
fi
#define VerifyFreshness(cr,ex) if
:: ((cr == CREATED)&&(ex == EXPIRES))→ valid_TimeStamp =1
:: else skip
fi
```

STP Global Variables

```
bit SenderChallengeA_STS=0;
bit SenderBindA_STS=0;
bit RecvrChallengeA_STS=0;
bit RecvrBindA_STS=0;
bit SenderChallengeI_STS=0;
bit SenderBindI_STS=0;
bit RecvrChallengeI_STS=0;
bit RecvrBindI_STS=0;
bit MsgType_RST = 0;
bit MsgType_RSTR = 0;
bit valid_dsSigA = 0;
bit valid_dsSigSTS = 0;
bit valid_dsSigI = 0;
bit partial_SC = 0;
bit Security_Context=0;
bit Security_Context_I=0;
int isEncrypted = 0;
bit valid_TimeStamp =0;
bit Imposter_FLAG = 0;
bit valid_DecryptA = 0;
bit valid_DecryptSTS = 0;
bit valid_DecryptI =0;
```


CARDIFF UNIVERSITY
PRIFYSGOL CAERDYDD