Automated analysis of XML-based protocols

Morten Barklund

Kongens Lyngby 2007
IMM-Thesis-2007-44

Technical University of Denmark
Informatics and Mathematical Modelling
Building 321, DK-2800 Kongens Lyngby, Denmark
Phone +45 45253351, Fax +45 45882673
reception@imm.dtu.dk
www.imm.dtu.dk

Summary

Static protocol analysis has always focused on formal protocols specified by standardisation organisations, researchers and other "professionals". Custom protocols created by software developers around the world needing network-based interfaces between systems have rarely been the topic for protocol analysis, as developers rarely specify them in a manner suitable for formal methods, nor do they have access to the somewhat academic means by which, they could perform this analysis.

One of the emerging standards for developer-created protocols (known as Web services) is SOAP[21]. SOAP is an XML-based protocol originally developed for remote method invocation, but since then developed far beyond original intent and can among other things contain complex authorization schemes. Thus, a specification language for specifying the security of SOAP protocols have been established — Web Services Enhancements[17].

This enhancement was originally created by Microsoft, but has had ammendments created by other key stakeholders as well. It is a highly pluggable architecture (as XML often is) with many different applications and intentions.

The purpose of this report is to derive a formal protocol from a web service specification in a manner suitable for static validation of the security of the protocol. This will utilize LySatool[16] and the foundations of static protocol validation as outlined by H. Nielson et al.[3].

The first chapter will provide sufficient background material for the technologies used necessary for understanding the scope of the project. The second chapter

will go in-depth with the understanding and transformation of the protocol in theory. Third chapter is an implementation of this discussing more practical challenges and solutions.

Thr fourth chapter concerns the results and applications of this project — how and why is this interesting. Finally, the conclusion will reflect upon the results as well as give an updated discussion of the current state of Web Service technologies and their applications.

Resumé

Statisk protokolanalyse har altid fokuseret på formelle protokoller specificeret af standardiseringsorganer, forskere eller andre "professionelle". Arbitrære protokoller defineret af softwareudviklere rundt om i verden med behov for netværksbaserede grænseflader mellem systemer har sjældent været målet for protokolanalyse, da de sjældent specificeres på en passende måde, og samtidig har udviklere sjældent viden om eller adgang til de akademiske metoder med hvilke, de kunne udføre denne analyse.

En af de større standarder for udvikler-skabte protokoller (kendt som Webservices) er SOAP[21]. SOAP er en XML-baseret protokol oprindeligt udviklet til fjern-metode-invokering (eng.: *remote method invocation*), men er siden udviklet langt udover den oprindelge intention og kan blandt andet indeholde komplekse autorisationsmetoder. Derfor blev et specifikationssprog udviklet til at specificere en SOAP-protokols sikkerhedskrav — Web Services Enhancements[17].

Denne udvidelse var oprindeligt udviklet af Microsoft, men har siden også fået tilføjelser fra andre nøgleinteressenter. Det er en meget modulær arkitektur (som XML ofte er) med mange applikationsmuligheder og formål.

Formålet med denne rapport er at udtrække en formel protokol fra en webservice-specifikation til et format, der kan underkastes statisk protokolanalyse. Dette vil basere sig på LySatool[16] og det fundament for statisk protokolanalyse lagt af H. Nielson et al.[3].

Første kapitel giver det fornødne baggrundsmateriale for de teknologier, som er nødvendige for forståelsen af projektet. Andet kapitel går i dybden med

forståelsen for og teorien bag konverteringen af protokollen. Tredje kapitel beskriver implementationen af denne konvertering samt mere praktiske udfordringer og løsninger.

Fjerde kapitel omhandler resultaterne og anvendelserne af dette projekt — hvor og hvorfor er dette interessant. Til sidst vil konklusionen reflektere over de opnåede resultater såvel som give en opdateret diskussion af Webserviceteknologiernes nuværende stadier og deres anvendelser.

Preface

This thesis was prepared at Institute for Informatics and Mathematical Modelling at the Technical University of Denmark in fulfillment of the requirements for acquiring a Master of Engineering.

This thesis documents my work on automated validation of XML-based protocols done during the winter and spring of 2007.

The work primarily focuses on SOAP and the many off-spring "standards" coined from this specification. The work tries to convert the many soft specifications of SOAP message security to the proven rules for input to LySatool — an automated security protocol validator.

Lyngby, June 2007 Morten Barklund

Acknowledgements

I thank my advisor, Hanne Riis Nielson at Institute for Mathematics and Modelling at Technical Univerisy of Denmark - IMM at DTU. She helped me to identify relevant predecessors of the topic as well as guide me in the direction of the scope and reflective level of this report.

Furthermore, I would of course like to thank my employer and colleagues for giving me the time and support needed to complete this report in a more than acceptable way whilst being under the pressure of a full-time job.

Contents

Chapter 1

Domain analysis

In order to understand the scope, intents and results of this project properly, a certain background analysis is necessary. This chapter will provide not only descriptions, but attempt to summarize and formalize information as well.

The elements covered here is first and foremost SOAP and web services. A brief history, a summarization of key elements and formal descriptions of Web Service specification.

Second, the whole concept of web service validation is targeted. For what reasons and purposes is validation performed. What will come from proper static validation of a web service — and what could come from lack thereof? This will also address a tool created by Microsoft Research Project Samoa[18], Tulafale[20] and it's approach to protocol validation.

Third, the syntax and semantics of LySatool will be introduced. This will be followed by an introduction to the conversion from Web Service Enhancement protocol syntax to LySatool input syntax as well as interpreting the LySatool result with respect to the given protocol.

Finally, the application of this project will be layed out with respect to architecture and distribution.

1.1 SOAP and web services

SOAP has gone through a turbulent evolution in a short time and has had high attention from many stakeholders driving it's widespread use even faster and further. Especially Microsoft has been one of the main agitators for this protocol.

1.1.1 History

Originally SOAP stood for *Simple Object Access Protocol* and was created as an alternative to Corba, because Corba used a port most often blocked by firewalls and secondly used a non-readable binary format.

1.1.1.1 Version 1.1

The initial Simple Object Access Protocol (SOAP) 1.1 note[5] was sent to W3C in 2000 and preceeded (and urged) the formation of a W3C working group in the area of XML-based protocols. SOAP was designed as a transport protocol meant to be sent on the application layer — an idea in itself against the OSI line-of-thought, and thus raised many eyebrows back then and still does.

The key points of this note (meant as a specification draft submitted by Microsoft, IBM, DevelopMentor, Lotus and UserLand) was to create a simple packaging system as a "lightweight protocol for exchange of information in a decentralized, distributed environment". It specified how messages consisted of an adressing header and a body containing typed arguments — and how the response to such was constructed similarly for both good and bad results (*faults*).

Furthermore the note did not require any one (application) tunnelling protocol for this transport protocol to work through, but outlined how tunnelling should be performed as well as gave a recommendation as to how HTTP could be used as tunnel.

The note also lay the fundation as to how SOAP is not a client-server protocol, but in stead should be seen as a SOAP *message chain* (as other transport protocols), in which at each level the SOAP *intermediary* will examine the adressing headers for this intermediary (indicated by the *role* or *actor* of the header entry) and act correspondingly. The idea is, that any intermediary can receive any

message and simply pass it along to(wards) the right *ultimate receiver* without prior knowledge of the actual application requesting or being invoked.

1.1.1.2 Version 1.2

The W3C working group formed after the submission of the above note finally published SOAP version 1.2, in which SOAP was not an acronym anymore, but just a name. Primarily because it was no longer confined to *object access* — nor was it *simple*.

SOAP version 1.2 is divided into three parts:

- Part 0: Primer[19] — use cases introducing SOAP version 1.2 in normative text.
- Part 1: Messaging Framework[13] — the basic SOAP version 1.2 packaging format and underlying protocol requirements.
- Part 2: Adjuncts[14] — further extensions of the messaging framework.

The interesting part with respect to this report is part 1 about the messaging framework. It reformulates and to some degree extends the suggestions put forward in the 1.1 note. Especially the message chain methodology is expanded upon describing in detail (though informally) how messages should be handled by intermediary and ultimate receivers.

The key differences (besides changes in choice of words) between the messaging framework as presented in the SOAP 1.2 specification and the original proposal in the SOAP 1.1 note can be summarized as:

- The SOAP technology is made more transport-neutral and removes all things remotely related to the suggested http transport protocol (though http is still suggested as a transport protocol — but not intertwined with the SOAP processing model).
- The descriptions of syntax. SOAP 1.1 used the XML specification[6] and it's tight EBNF-rules for describing elements and attributes. SOAP 1.2 uses the Infoset specification[11], which is a very "loose" specification only describing intent — not actual implementation. This is by many considered a bad choice as it makes implementation harder and it opens for different interpretations of the infoset specification. For example, legal

characters in element node names is not specified at all in the infoset specification, whereas in the XML specification, it is specified down to the last unicode character position.

Nonetheless, SOAP 1.2 is much more formal and concise that the 1.1 note. The ideas and extent of the protocol however remain somewhat the same.

1.1.1.3 Web Service Description Language (WSDL)

As the above tells, SOAP became more and more complex, and what was meant as (and based on) humanly readable syntax, became too complex for regular developers to worry about. Thus came the Web Service Description Language[22], which is still XML-based, but enables application developers to use a tool to create a Web Service Description Language file, a WSDL (often pronounced "wiz-dull") that could be sent to other developers. Though XML-based, it was a language meant for automatic generation of web service endpoints. For instance a service provider would distribute a WSDL file to clients, that could then automatically generate the set of functions needed to invoke this web service in the language of choice — be it Java, .NET or something third.

WSDL is as of currently not ratified as a specification by any standardization organisation. Version 2 exists in many drafted versions however, and the latest is expected to become a W3C recommendation.

The development of WSDL resembles SOAP in a peculiar way — thus an initial note by Microsoft and IBM suggesting WSDL 1.1[10] was released in 2001 and since then a set of proposals in three parts were submitted to W3C for recommendation as version 2.0 in 2007:

- Part 0: Primer[4] — use cases introducing WSDL version 2 in normative text.
- Part 1: Core[9] — the basic contents of WSDL specification, distribution and interpretation.
- Part 2: Adjuncts[8] — further extensions of the core.

WSDL however, is an extension to make SOAP more easily implementable, but it does not provide security improvements, nor does it specify policies.

Therefore, WSDL in it's current unrecommended state does not influence this project — and it is worth noting, that it is still under development. Latest version of the above 3 documents came as late as May 23rd 2007.

1.1.2 Web Services Enhancement (WSE)

Web Services Enhancements (WSE)[17] is a technology produced by Microsoft in several versions. The fundamental idea is to produce a common set of functions for checking that certain standardized security measures are in place. It is a set of filters that developers can utilize for SOAP nodes sending and/or receiving SOAP messages.

WSE both consists of a specification describing security policies for sending and receiving messages as well as a set of classes for C# development providing common functionality needed to enforce common security measures that can be specified by the accompanying specification.

One of the central aspects of WSE is however somewhat against the original intent of SOAP. The SOAP message chain is not considered that important any more, but direct client-server architecture is the corner stone of the ideas in WSE. The policy document specification however, is quite flexible and is able to (by proper interpretation) specify chained message policies trying to obey the message processing rules of SOAP 1.2.

The strangest part about WSE is, that it is merely a specification not related to implementation. Developers specify the necessary web service security via this Web Services Enhancement policy document, and then they implement trying to uphold the given policy. But there is no way to actually check, that the implementation adheres the policy.

Originally there was no way to ensure, that the developed policy was sane. But for this, Microsoft Research had project Samoa, which will be described later. This research project among other things resulted in tools to check a policy for common flaws.

1.1.3 Formal syntax

In this subsection, I will try to formalise the syntax for SOAP messages, outline how they should be parsed by a SOAP node, and then outline the format of SOAP policies (WSE Policy Documents) and their interpretation.

No such formalisation exist as of currently. XML DTDs (Document Type Definitions) exist for both SOAP and policy documents, but DTDs only describe syntax, not intent — and DTDs describe syntax in an excessively verbose manner.

1.1.3.1 SOAP message format

The SOAP message format is build from the envelope, containing header and body. Headers contain *header blocks* (or *header entries* as they were called in 1.1):

$$
\begin{aligned}
\mathit{envelope} &:= \texttt{'<Envelope>'} \ , \ \mathit{header} \ , \ \mathit{body} \ , \ \texttt{'</Envelope>'} \\
\mathit{header} &:= \texttt{'<Header>'} \ , \ \{ \ \mathit{headerBlock} \ \} \ , \ \texttt{'</Header>'} \\
\mathit{envelope} &:= \texttt{'<Body>'} \ , \ \mathit{string} \ , \ \texttt{'</Body>'} \\
\mathit{headerBlock} &:= \texttt{'<'} \ , \ \mathit{string} \ , \ \texttt{' '} \ , \ \{ \ \mathit{attribute} \ \} \ , \ \texttt{'>'} \ , \\
&\quad\ \ \mathit{string} \ , \ \texttt{'</'} \ , \ \mathit{string} \ , \ \texttt{'>'} \\
\mathit{attribute} &:= \mathit{string} \ , \ \texttt{'="'} \ , \ \mathit{string} \ , \ \texttt{'" '}
\end{aligned}
$$

That is, the required basic format is quite simple. And the basic rules for parsing this consist of the rules as specified in SOAP 1.2: Message Format section 2.6: *Processing SOAP Messages*:

1. Determine the set of roles in which the node is to act. The contents of the SOAP envelope, including any SOAP header blocks and the SOAP body, MAY be inspected in making such determination.
2. Identify all header blocks targeted at the node that are mandatory.
3. If one or more of the SOAP header blocks identified in the preceding step are not understood by the node then generate a single SOAP fault with the *Value of Code* set to ‘‘`env:MustUnderstand`’’. If such a fault is generated, any further processing MUST NOT be done. Faults relating to the contents of the SOAP body MUST NOT be generated in this step.
4. Process all mandatory SOAP header blocks targeted at the node and, in the case of an ultimate SOAP receiver, the SOAP body. A SOAP node MAY also choose to process non-mandatory SOAP header blocks targeted at it.
5. In the case of a SOAP intermediary, and where the SOAP message exchange pattern and results of processing (e.g. no fault generated) require

> that the SOAP message be sent further along the SOAP message path, relay the message as described in section 2.7 Relaying SOAP Messages.

The first point in the above is a bit tricky. There is actually a point before that, which is kept quite secret in the specification. That is, how does the SOAP node determine, who it is? If for instance message is sent with a header block specififying:

```
<To>http://domain.invalid/service</To>
```

How does the node know, if the node matches this URL? Because if it does, the node is to act as the *ultimate receiver*, but if not, it is only an intermediary node. This resolution of node identity is covered in a note in section 2.2:

> For example, implementations can base this determination on factors including, but not limited to: hard coded choices in the implementation, information provided by the underlying protocol binding (e.g. the URI to which the message was physically delivered), or configuration information provided by users during system installation.

This gives an inconsistency in the specification. When the specification says, that any SOAP node must be able to act as an intermediary node without prior knowledge about the application protocol being transfered in the SOAP messages, then the node should be able to directly identify, whether it is the ultimate receiver of the message or not. And if not, determine which if any header blocks in the message is meant for the node. But as a header block directly identifying `<To>` is not required, then any node cannot extract who the ultimate receiver of the message is directly from any SOAP message — and thus not know, if the node itself is the ultimate receiver.

In order to formalise this, I need a unique way to identify the ultimate receiver of a message, so any node can determine, if it has this property. And then I need a unique way to name nodes, so any node immediately knows it's own name. For this use, I will choose the network name of the node as the identity of the node as well as require a `<To>` header block identifying the ultimate receiver of the message.

This enables me to give this more formal procedure written in pseudo-code using XPath-like traversing of the received document:

```
HANDLE-SOAP-MESSAGE(self, message)
 1  to ← XPATH(message, /envelope/header/to[0])
 2  if to = self
 3     then role ← NEXT
 4     else  role ← ULTIMATERECEIVER
 5  headers ← XPATH(message, /envelope/header[role = role|role = self])
 6  if role = ULTIMATERECEIVER
 7     then headers ← headers ⋃
 8               XPATH(message, /envelope/header[role =])
 9  for  each header in headers
10  do result = HANDLEHEADER(header)
11       mustUnderstand ← XPATH(header, [@mustUnderstand])
12       if mustUnderstand ≠ NIL ∧ result = NIL
13          then return FAULT(header)
14  if role = ULTIMATERECEIVER
15     then body ← XPATH(message, /envelope/body)
16             return HANDLEBODY(headers, body)
17     else  return FORWARDMESSAGE(message, to)
```

The auxilary functions for handling body and headers would then be application specific. And the methods for forwarding the message (and waiting for the response to return) as well as the fault generation method would be global (but trivial).

1.1.3.2 WSE Policy Document format

The Poliy Document format described in Web Services Enhancements is build on the following general structure:

$$
\begin{aligned}
\textit{policyDocument} &:= \texttt{'<policyDocument>'} , \textit{mappings} , \textit{policies} , \\
&\quad \texttt{'</policyDocument>'} \\
\textit{mappings} &:= \texttt{'<mappings>'} , \{ \textit{endpoint} \} , [\textit{defaultEndpoint}] , \\
&\quad \texttt{'</mappings>'} \\
\textit{endpoint} &:= \texttt{'<endpoint uri="'} , \textit{string} , \texttt{'">'} , \\
&\quad \textit{endpointDescription} , \texttt{'</endpoint>'} \\
\textit{defaultEndpoint} &:= \texttt{'<defaultEndpoint>'} , \textit{endpointDescription} , \\
&\quad \texttt{'</defaultEndpoint>'} \\
\textit{endpointDescription} &:= \{ \textit{operation} \} , \textit{defaultOperation} \\
\textit{operation} &:= \texttt{'<operation request="'} , \textit{string} , \texttt{'">'} , \\
&\quad \textit{operationDescription} , \texttt{'</operation>'} \\
\textit{defaultOperation} &:= \texttt{'<defaultOperation>'} , \textit{operationDescription} , \\
&\quad \texttt{'</defaultOperation>'} \\
\textit{operationDescription} &:= [\textit{request}] , [\textit{response}] \\
\textit{request} &:= \texttt{'<request policy="'} , \textit{string} , \texttt{'" />'} \\
\textit{response} &:= \texttt{'<response policy="'} , \textit{string} , \texttt{'" />'} \\
\textit{policies} &:= \texttt{'<policies>'} , \{ \textit{policy} \} , \texttt{'</policies>'} \\
\textit{policy} &:= \texttt{'<Policy Id="'} , \textit{string} , \texttt{'">'} , \textit{policyReq} , \\
&\quad \texttt{'</Policy>'} \\
\textit{policyReq} &:= \texttt{'<all>'} , \{ \textit{policyReq} \} , \texttt{'</all>'} \mid \\
&\quad \texttt{'<oneormore>'} , \{ \textit{policyReq} \} , \texttt{'</oneormore>'} \mid \\
&\quad \texttt{'<exactlyone>'} , \{ \textit{policyReq} \} , \texttt{'</exactlyone>'} \mid \\
&\quad \textit{integrity} \mid \textit{confidentiality}
\end{aligned}
$$

This assigns unique id's to policies and then assigns policies (by id's) to operations of endpoints.

And then comes the confidentiality node:

$$
\begin{aligned}
confidentiality &:= \texttt{'<Confidentiality>'} , [\, keyInfo \,] , confMessageParts , \\
&\quad \texttt{'</Confidentiality>'} \\
keyInfo &:= \texttt{'<KeyInfo>'} , securityToken , [\, securityTokenRef \,] , \\
&\quad \texttt{'</KeyInfo>'} \\
securityToken &:= \texttt{'<SecurityToken>'} , [\, tokenType \,] , [\, claims \,] , \\
&\quad \texttt{'</SecurityToken>'} \\
tokenType &:= \texttt{'<TokenType>'} , string , \\
&\quad \texttt{'</TokenType>'} \\
claims &:= \texttt{'<Claims>'} , \{\, claim \,\} , \texttt{'</Claims>'} \\
claim &:= subjectName \mid role \\
subjectName &:= \texttt{'<SubjectName>'} , string , \texttt{'</SubjectName>'} \\
role &:= \texttt{'<Role value="'} , string , \texttt{'" />'} \\
securityTokenRef &:= \texttt{'<SecurityTokenReference>'} , keyIdentifier , \\
&\quad \texttt{'</SecurityTokenReference>'} \\
keyIdentifier &:= \texttt{'<KeyIdentifier>'} , string , \\
&\quad \texttt{'</KeyIdentifier>'} \\
confMessageParts &:= \texttt{'<MessageParts>'} , string , \\
&\quad \texttt{'</MessageParts>'}
\end{aligned}
$$

And the integrity node (sharing many rules from the confidentiality node):

$$
\begin{aligned}
integrity &:= \texttt{'<Integrity>'} , [\, tokenInfo \,] , intMessageParts , \\
&\quad \texttt{'</Integrity>'} \\
tokenInfo &:= \texttt{'<TokenInfo>'} , securityToken , \\
&\quad \texttt{'</TokenInfo>'} \\
intMessageParts &:= \texttt{'<MessageParts>'} , string , \\
&\quad \texttt{'</MessageParts>'}
\end{aligned}
$$

The difference between the two types of message parts is not clear from the above — but in the implementation, one would of course never (be able to) require confidential header blocks, as they must be readable to any node. Only

the body of the message can be confidential. Integrity however, can be claimed on any set of header blocks and/or message body.

Interpreting this however, is quite complex. Interpretation is only given via a set of small examples in the documentation and some general notes to some of the nodes.

But from the specification, one can try to derive the following procedure for handling SOAP messages with respect to this policy (in fact, this would be some of the things to do in the `HandleHeader` method invoked in the previous algorithm). When a SOAP message has arrived to a node, the node should check SOAP message security with respect to the policy document as:

1. Determine the endpoint and operation the message has been sent to and tries to invoke.
2. Find the request policy id in the policy document for the given endpoint and operation — and use default endpoint and/or operation if necessary.
3. If the request policy id is empty, return with success.
4. If the request policy id refers to a non-existent policy within the policy document, return with fatal error.
5. Otherwise, check the policy requirements with respect to the boolean operators, that enclosing quantifiers represent:
   (a) If within `all`, all requirements must be fulfilled
   (b) If within `oneormore`, at least one requirement must be fulfilled
   (c) If within `exactlyone`, exactly one requirement must be fulfilled
6. For integrity requirements, examine the security token if exists.
   (a) If token type is specified, check whether to use username-password or digital signature.
   (b) If claims are specified, for each claim check that it holds
      i. If a subject name claim is given and the token type is username-password, check that the username matches the claimed username.
      ii. If a subject name claim is given and the token type is digital signature, check that the signing entity matches the claimed username.
      iii. If a role claim is given check that the username or signing entity has this role (and the privileges of this role) within some local lookup

(c) If any claim fails, return false.

(d) If all given claims hold, match the header block containing the check value against the given token type

i. If token type is username-password, check that the password along with the given message parts hashed (using an agreed algorithm like SHA-1) matches the received check value

ii. If token type is digital signature, check that the received check value combined with the public key corresponding to the digital signature corresponds with the hash value obtained from hashing the specified message parts.

7. For confidentiality requirements, examine the security token if exists.

(a) If token type is specified, check whether to use username-password or public-key encryption.

(b) If claims are specified, for each claim check that it holds

i. If a subject name claim is given and the token type is username-password, check that the username matches the claimed username.

ii. If a subject name claim is given and the token type is public-key encryption, check that the public key belongs to the claimed username.

iii. If a role claim is given check that the username or public key owner has this role (and the privileges of this role) within some local lookup

(c) If any claim fails, return false.

(d) If all given claims hold, decrypt the message body.

i. If token type is username-password, decrypt the message body using the secret password corresponding to the user.

ii. If token type is public-key encryption, decrypt using the private key completing the public key.

This establishes a non-deterministic algorithm, as many policy requirements within a `oneormore` node can be checked in any order and if any of these hold, then the rest need not be checked. In practive however, the quantifying nodes will not be used very excessively. There are only a few combinations of policy requirements, that make sense. Most will follow a simple pattern: require that one of a set of potential integrity methods have been uphold and/or require, that the body is encrypted in a certain way. It does not make sense to require, that either is the message signed or the body is encrypted, as these two things serve two different purposes and thus cannot be interchanged.

This does not easily translate to pseudocode enlightning the algorithm better, than the above procedure description — there are simply too many trivial tasks involved.

1.2 Validation of web-services

Protocol security primarily has two purposes:

- The receiver needs to be sure, that sender is, who he says he is — *integrity*
- The sender needs to be sure, that noone but the intended receiver can read his message — *confidentiality*

And the exact same things applies to SOAP and web services. Some web services need not bother about any of these — for example an interface to Google Search. Google has a public SOAP interface, so any website can show a search field and display the results in their own manner — but in the background use Google's web service. If someone intercepts the information transfered, it is not a huge security breach, and if someone injects bad results, it is a mere nuisance.

Some services needs to make sure, that both properties are maintained — both for the request sent from the client and for the response sent by the server. This could for example be an interface to social security information accessible by other governmental agencies. Here, both the request and the response can contain secret information and thus confidentiality is needed. But it is also required, that only the proper agencies can request information as well as these agencies needs to be sure, that the correct web service sent the response, and thus integrity is needed as well.

For SOAP these two issues are approached in the same manner as in most other protocols. Integrity is implemented through either digital signatures or one-way-hashing using username-password. Confidentiality is either achieved using shared private keys or private-public key pairs. As the SOAP protocol specifies, that any SOAP intermediary must be able to handle the message, only the message body can be encrypted though. The message header must remain in clear text. The integrity is most often embedded as a signature in a header block.

1.2.1 Protocol attacks

Potential simple protocol attacks on SOAP security would be for instance a situation, where a sender digitally signs his message, but he simply signs uninteresting attributes — for example *recipient* only. Even though the attacker cannot recreate this signature, he can simply reuse the same signature and send another message to the same recipient pretending to be the original sender.

To analyse potential attacks on protocols, Dolev and Yao[12] has created a notion of the *hardest attacker*. The Dolev-Yao model specifies what an attacker can do to suit his purpose (guess secrets): the intruder can intercept any message, forge new messages and send them using an honest agent's identity.

1.2.2 Microsoft Research Project Samoa

Due to the problem of developer-specified policies, that might not be secure (given that some developers working with web services might not have the proper background), Microsoft Research launched project Samoa with the purpose of enhancing Web Services Enhancements

From their description, one can read that:

> The underlying principles, and indeed the difficulties, of using cryptography to secure RPC protocols have been known for many years, and there has been a sustained and successful effort to devise formal methods for specifying and verifying the security goals of such protocols. One line of work, embodied in the spi calculus of Abadi and Gordon and the applied pi calculus of Abadi and Fournet, has been to represent protocols as symbolic processes, and to apply techniques from the theory of the pi calculus, including equational reasoning, type-checking, and resolution theorem-proving, to attempt to verify security properties such as confidentiality and authenticity, or to uncover bugs.

This project released several tools during 2005. First, they released a *WSE Policy Advisor* for WSE 2.0 and 3.0, which are program extensions, that developers could install along with their WSE installations. This would then give the option, that given a policy, one could check it for known flaws, potential security attacks and common pitfalls.

Furthermore, Microsoft released the underlying engine, Tulafale, which is the actual program performing these security checks. It is based on the pi calculus and Blanchet's resolution-based protocol verifier[2].

Accompanying Tulafale is a set of papers describing the development of Tulafale. There are several problems with these papers though. First of all, Tulafale does not work directly on WSE Policy Documents. It works on an XML-format close to the WSE Policy Document model, but not exactly the same. This seems very strange, but it probably has to do with the fact, that the Policy Advisors are coupled very closely with the development environments, that developers use — and as such the Policy Advisor can facilitate on an internal representation of the Policy Document in stead of the *real* document.

This however leaves this project with the disadvantage, that if we want to mimick Tulafale, we must be able to parse their non-validating XML-documents. And if we want to parse the real Policy Document standard, we must parse a document, that Tulafale does not understand.

One of the simplest problems with Tulafale is the outer elements — Tulafale input, *policyMappings*, is defined as:

$$
\begin{array}{rcl}
\mathit{policyMappings} & := & \texttt{'<PolicyMappings>'} \ , \ \{ \ \mathit{genericPolicy} \ \} \ , \\
 & & \texttt{'</PolicyMappings>'} \\
\mathit{genericPolicy} & := & \mathit{sendPolicy} \mid \mathit{receivePolicy} \\
\mathit{sendPolicy} & := & \texttt{'<SendPolicy>'} \ , \ \mathit{policyDescription} \ , \\
 & & \texttt{'</SendPolicy>'} \\
\mathit{receivePolicy} & := & \texttt{'<ReceivePolicy>'} \ , \ \mathit{policyDescription} \ , \\
 & & \texttt{'</ReceivePolicy>'} \\
\mathit{policyDescription} & := & \mathit{to} \ , \ \mathit{action} \ , \ \mathit{policy} \\
\mathit{to} & := & \texttt{'<To>'} \ , \ \mathit{string} \ , \ \texttt{'</To>'} \\
\mathit{action} & := & \texttt{'<Action>'} \ , \ \mathit{string} \ , \ \texttt{'</Action>'}
\end{array}
$$

Then in the above, *policy* matches the proper format for describing policies. Thus, where Policy Documents describe a slightly different but equally powerful markup, Tulafale uses the above format. The reason for this is unknown.

Some of the examples provided with Tulafale does not even adhere to the structure of Policy Documents within the policies either. For example, Tulafale accepts a `<Confidentiality>` node with a `<TokenInfo>` node beneath it — confidentialty-nodes are by the specification only accompanied with a `<KeyInfo>` node. The difference between `<TokenInfo>` and `<KeyInfo>` is, that for the key

info node, the policy can require — within a `<SecurityTokenReference>` node — which public key is to be used to encrypt the contents (which is implicitly required, as only the correct public key will make the content decryptable).

The reason for this is also unknown, but one guess is, that as Tulafale does not support the security token reference, the difference between token info and key info is negligable.

This report will work focus only on the proper policy documents described by WSE, but any policy mapping used for Tulafale could easily be rewritten to a proper policy document and thus be validated.

1.3 LySatool

LySa is a process calculus and LySatool provides a security analysis based on the calculus. LySatool accepts protocols in a notation allowing for several simple network messaging constructs like sending and receiving messages over one, common network channel as well as cryptographic primitives for encryption and decryption using either shared secret or private-public key pairs.

1.3.1 Syntax

The syntax is very simple and layed out in a formal, specific manner in the user guide[7] and thus will not be presented here.

The main ingredients of LySatool input is processes, *proc*, and terms, *term*.

Processes are the root component of all LySatool inputs and is a recursive structure allowing among other things for composition, concurrency and message passing and acceptance. Processes are non-deterministic (given the concurrency syntax).

Elements sent and accepted over the network are terms. Terms can be either variables, or secret-encrypted or public-key-encrypted terms.

The actual syntax will be given more thought when considering the translation of policy documents to LySatool input.

1.4 Intended use

The result of this project is a piece of software, that developers can invoke with a given policy document created by the definitions and tools in WSE, that can then be automatically validated with LySatool, and if potential protocol attacks exist, report what part of the policy document, that introduces these flaws.

The architecture will a web-based interface written in Python. This will then accept input policy documents, process these through a policy-document-to-lysa-converter, run LySatool on the result, interpret LySatool result with respect to the policy document and return these findings.

Chapter 2

Theory

This chapter will go into detail about how the translation from policy documents to LySatool input will be performed.

The parsing of policy documents is straight-forward and almost trivial, but the generation of LySatool input is far more complex. LySatool input requires that the actually sent elements are specified and in order to have two processes communicate via a shared secret, they need to specify exactly who the other process is in order to use the correct shared secret.

Thus this translation will be given much thought. This is done by first establishing how nodes can be converted to processes and what message path patterns have as consequence for these processes. Secondly, a formal method for security agreement is created in order to match security requirements of interacting nodes. And finally, the creation of processes is dealt with considering the initial foundation.

2.1 Internal representation

In order to translate the policy document, it will first be translated to an internal representation representing the intent of LySatool input — but not the

syntax. As a LySatool proces cannot use conditions on term values, the different choices of potential policies applicable to a given endpoint will be implemented as concurrent processes in LySatool input (for policy requirements of which only one is required to hold) or composite processes (for policy requirements of which all are required to hold).

This internal representation is based around endpoints. As LySatool input normally is constructed with a set of concurrently running processes each specifying a stakeholder, each endpoint can be considered a stakeholder. For each operation each stakeholder (endpoint) has, a process must be generated. And for each operation of each stakeholder, and anonymous client process must be set up, that can call this operation in order to test the values sent over the network. And each operation will have a request and a response policy. Either or both of these policies can be null, which means that no restrictions will be imposed on the values sent or received respectively.

If a policy requires integrity, one or more integrity schemes can be required and corresponding check values must thus be sent satisfying the required set of integrity schemes.

If a policy requires confidentiality, one (of a number of) encryption methods will be required and corresponding message body encryption must be applied satisfying the required encryption methods.

A model consists of a set of stakeholders, and each stakeholder has a policy for incoming messages and a policy for outgoing messages:

$$\begin{aligned} model &:= \{\ stakeholder\ \} \\ stakeholder &:= policy\ ,\ policy \end{aligned}$$

A policy consists of a message (either for requests or responses), a confidentiality policy set for the message and an integrity policy set for the message:

$$policy := message\ ,\ confPolicySet\ ,\ intPolicySet$$

The message contains a set of header entries — each of these being the name of the header — and a maybe a message body — indicated by a simple flag:

$$
\begin{aligned}
message &:= \{\ header\ \}\ ,\ body \\
header &:= string \\
body &:= \texttt{HASBODY}|\texttt{HASNOBODY}
\end{aligned}
$$

The two policy sets are almost equivalent in that they both have a combinator — from the (possibly empty) set of policies, either all should be valid, any should be valid or exactly one should be valid. If no policies exist in the set, the policy set is always valid without concern for the combinator.

$$
\begin{aligned}
intPolicySet &:= combinator\ ,\ \{\ intPolicy\ \} \\
confPolicySet &:= combinator\ ,\ \{\ confPolicy\ \} \\
combinator &:= \texttt{ALL} \mid \texttt{ONEORMORE} \mid \texttt{EXACTLYONE}
\end{aligned}
$$

Confidentiality policies has been described several times, but for this internal representation they consist of a potential token type (which could include a public key) and a set of claims:

$$
\begin{aligned}
confPolicy &:= [\ confTokenType\ ]\ ,\ \{\ claim\} \\
confTokenType &:= \texttt{SECRET} \mid \texttt{PUBLICKEY}\ ,\ string \\
claim &:= \texttt{SUBJECT}\ ,\ string \mid \texttt{ROLE}\ ,\ string
\end{aligned}
$$

In this definition it is assumed obvious, that the confidential part of the message is the body and the body alone.

The integrity policy is almost the same, in the way that it defines a token type, a set of claims but also a set of message parts:

$$
\begin{aligned}
intPolicy &:= [\ intTokenType\ ]\ ,\ \{\ claim\}\ ,\ \{\ part\ \} \\
intTokenType &:= \texttt{SECRET} \mid \texttt{SIGNATURE} \\
part &:= \texttt{BODY} \mid \texttt{HEADER}\ ,\ string
\end{aligned}
$$

And this completes the internal representation of policy documents - which is a state somewhere between expressing the power of policy documents and the power of LySatool input.

2.2 Parsing policy documents to internal representation

In order to parse policy documents (proper documents as outlined by the WSE specification and not as accepted by Tulafale), we have to define a mapping from the XML document tree to the internal representation. This is done as a set of functions from an XML tree to the individual parts of the internal representation.

The outermost function takes a policy document XML tree and returns a model:

```
TranslatePolicyDocument(policyDocument)
 1  policies ← TranslatePolicies(XPath(policyDocument, /policies))
 2  endpoints ← XPath(policyDocument, /mappings/∗)
 3  stakeholders ← ∅
 4  for each endpoint in endpoints
 5  do operations ← XPath(endpoint, /∗)
 6     for each operation in operations
 7     do receivePolicy ← policies[XPath(endpoint, /receive@id)]
 8        responsePolicy ← policies[XPath(endpoint, /response@id)]
 9        stakeholder ← StakeHolder(receivePolicy, responsePolicy)
10        stakeholders ← stakeholders ⋃ {stakeholder}
11
12 return new Model(stakeholders)
```

This sets up the stakeholders comprising the model by requiring a map from policy identifiers to policy constructions. This function is then done as:

```
TranslatePolicies(policies)
 1  policyMap ← ∅
 2  for each policyDescr in policies
 3  do headers ← ExtractHeaders(policyDescr)
 4     body ← ExtractBody(policyDescr)
 5     message ← new Message(headers, body)
 6     integrity ← ExtractIntegritySet(policyDescr)
 7     confidentiality ← ExtractConfidentialitySet(policyDescr)
 8     policy ← new Policy(message, integrity, confidentiality)
 9     policyId ← XPath(policyDescr, @id)
10     policyMap[policyId] ← policy
11  return policyMap
```

The two first extraction functions for finding reference header elements and message body are done as:

```
ExtractHeaders(policy)
1  headers ← ∅
2  messageParts ← XPath(policy, //messageparts/text())
3  for each messagePart in messageParts
4  do headers ← headers ⋃ Split(messagePart, ' ')
5  return headers
```

```
ExtractHeaders(policy)
1  bodyParts ← XPath(policy, //messageparts/[text()~='body'])
2  if bodyParts ≠ NIL
3    then return HASBODY
4  return HASNOBODY
```

The function for extracting the confidentiality set for a given policy is a bit more complex. Note that already here is made a minor optimization of input. It does not make sense (and is not possible) to require, that the body is encrypted in two different ways — though they could be on top of each other, but this will be disregarded as being plain silly — thus if several confidentiality policies co-exist, they must be wrapped in a `EXACTLYONE`, as only exactly one of these policies can (and must) be respected. If any other wrapping is in place, a warning will be thrown and `EXACTLYONE` will be used — and this warning will be part of the overall output. First the algorithm handles the two cases where zero or only one confidentiality policy requirement exists:

```
ExtractConfidentialitySet(requirements)
 1  combinator ← EXACTLYONE
 2  policySet ← ∅
 3  policies ← XPath(requirements, //confidentiality))
 4  switch
 5    case Length(policies) = 0 :
 6         return new ConfidentialitySet(policySet, combinator)
 7
 8    case Length(policies) = 1 :
 9         policySet ← {TranslateConfidentiality(policies[0])}
10         return new ConfidentialitySet(policySet, combinator)
11
12    case default :
13         morePolicies ← XPath(requirements, //oneormore/confidentiality)
14         if Length(morePolicies) > 0
15           then error "Only one confidentiality policy can be enforced"
16         allPolicies ← XPath(requirements, //all/confidentiality)
17         if Length(allPolicies) > 0
18           then error "Only one confidentiality policy can be enforced"
19         for  each policy in policies
20         do policySet ← policySet ⋃{TranslateConfidentiality(policy)}
21         return new ConfidentialitySet(policySet, combinator)
```

The actual translation of the individual confidentiality policies requires first to extract the type of token and then extract the claims. For this the constants `X509Token` and `UPToken` represent these values:

```
X509Token  :=  "http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-x509-token-profile-1.0#X509v3"
  UPToken  :=  "http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-username-token-profile-1.0#UsernameToken"
```

And then the translation is:

```
TranslateConfidentiality(policy)
 1  tokenType ← NIL
 2  tokenTypeNode ← Xpath(policy, /keyinfo/securitytoken/tokentype/text())
 3  if tokenTypeNode ≠ NIL
 4     then switch
 5             case tokenTypeNode = X509Token :
 6                  referenceNode ← Xpath(policy, //securitytokenreference/text())
 7                  if referenceNode ≠ NIL
 8                     then tokenType ← PUBLICKEY(referenceNode)
 9                     else  tokenType ← PUBLICKEY
10
11             case tokenTypeNode = UPToken :
12                  tokenType ← SECRET
13
14  claims ← ∅
15  subjectNameClaim ← Xpath(policy, /keyinfo/claims/subjectname/text())
16  if subjectNameClaim ≠ NIL
17     then claims ← claims ⋃{SUBJECT(subjectNameClaim)
18  roleClaim ← Xpath(policy, /keyinfo/claims/role@value)
19  if roleClaim ≠ NIL
20     then claims ← claims ⋃{ROLE(roleClaim)
21  return new Confidentiality(tokenType, claims)
```

The integrity policy set is a bit more complex, as they can be collected under a combinator of different kinds. This method will though require, that all integrity policies are joined under the same type of combinator. The specification allows for structures like:

```
<onormore>
    <all>
        <integrity ... />
        <integrity ... />
    </all>
    <exactlyone>
        <integrity ... />
        <integrity ... />
    </exactlyone>
</onormore>
```

Which is quite a complex structure, that in practice does not make sense. Either you have several policies, that all must be enforced, or you have several policies of which (at least) one must be enforced. Thus, if they exist under different

combinators, all the integrity policies will be assumed belonging to a one-or-more combinator in stead (and an error will be displayed):

```
ExtractIntegritySet(requirements)
 1  combinator ← nil
 2  policySet ← ∅
 3  policies ← XPath(requirements, //integrity))
 4  switch
 5    case Length(policies) = 0 :
 6         return new IntegritySet(policySet, ALL)
 7
 8    case Length(policies) = 1 :
 9         policySet ← {TranslateIntegrity(policies[0])}
10         return new IntegritySet(policySet, ALL)
11
12    case default :
13         morePolicies ← Length(XPath(requirements, //oneormore/integrity))
14         allPolicies ← Length(XPath(requirements, //all/integrity))
15         onePolicy ← Length(XPath(requirements, //exactlyone/integrity))
16         switch
17           case morePolicies = Length(policies) :
18                combinator ← ONEORMORE
19
20           case allPolicies = Length(policies) :
21                combinator ← ALL
22
23           case onePolicy = Length(policies) :
24                combinator ← EXACTLYONE
25
26           case default : combinator ← ONEORMORE
27                error "Different combinators used for integrity policies"
28         for each policy in policies
29         do policySet ← policySet ⋃ {TranslateIntegrity(policy)}
30         return new IntegritySet(policySet, combinator)
```

The translation for the integrity node follows the translation of the confidentiality node without the public key and with the addition of the message parts:

```
TranslateIntegrity(policy)
 1  tokenType ← NIL
 2  tokenTypeNode ← Xpath(policy, /tokeninfo/securitytoken/tokentype/text())
 3  if tokenTypeNode ≠ NIL
 4     then switch
 5             case tokenTypeNode = X509Token : tokenType ← SIGNATURE
 6
 7             case tokenTypeNode = UPToken : tokenType ← SECRET
 8
 9  claims ← ∅
10  subjectNameClaim ← Xpath(policy, /tokeninfo/claims/subjectname/text())
11  if subjectNameClaim ≠ NIL
12     then claims ← claims ⋃{SUBJECT(subjectNameClaim)
13  roleClaim ← Xpath(policy, /tokeninfo/claims/role@value)
14  if roleClaim ≠ NIL
15     then claims ← claims ⋃{ROLE(roleClaim)
16  parts ← ∅
17  partsString ← Xpath(policy, /messageparts/text())
18  if partsString ≠ NIL
19     then parts ← Split(partsString, ' ')
20  return new Integrity(tokenType, claims, parts)
```

2.3 Outputting internal representation as LySatool input

This second part of the transformation is to transform the internal representation to LySatool input. For this, one first has to establish all the potential message paths and create LySatool processes for each of these.

2.3.1 Message paths and processes

The key part of the SOAP message chain concept is, that it is not interesting or relevant, how the message gets to the receiver. The only interesting things are how the message is:

1. Initially sent from the sender,
2. Relayed via intermediary nodes, and

3. Received by the ultimate receiver.

If we have a policy document specifying several endpoints, then we have the following scenarios for the initial send:

1. The sender sends a message directly to any of the endpoints or
2. The sender sends a message to one of the endpoints via one of the other endpoints as a (first) intermediary node.

Similarly, the ultimate receiver (any of the endpoints) can receive a message in two ways:

1. The ultimate receiver receives a message directly from the sender or
2. The ultimate receiver receives a message from the sender via one of the other endpoints.

And finally, any of the endpoints can act as an intermediary node:.

1. An intermediary node can receive a message from any of the other endpoints or the client and forward it to any of the other endpoints or the client (but of course not back to the sender).

It is assumed, that the initial client sending the request expects a response and will thus also act as an ultimate receiver, but the initial client will not be used as an intermediary note — nor will the node, that he is trying to reach. The request and the response need not follow the same path, but the ultimate receiver will always send the response back to the penultimate node, that send the request to him.

The number of processes cannot be determined without knowing the policy requirements of the endpoints — as will be discussed further in the next section.

But if we assume "worst-case" — e.g. they all require secret-based integrity (which is the most complex, as the share a secret and must know, who sends or receives the message) — then the number of processes necessary to enable all message paths from a client to any of n endpoints via any set of intermediary nodes (including the empty set) can be calculated as:

1. n processes from the client directly to the ultimate recipient,

2. $n(n-1)$ processes from the client to the recipient via another endpoint,

3. 1 process at each of the n endpoints receiving the message directly from the client,

4. $n-1$ processes at each of the n endpoints receiving the message from the client via one of the other endpoints,

5. $n-1$ processes for each of the n intermediary endpoints receiving a message from the client and forwarding it to any of the other endpoints,

6. $n-1$ processes for each of the n intermediary endpoints receiving a message from another endpoint and forwarding it to the client, and finally

7. $(n-1)(n-2)$ processes for each of the n intermediary endpoints receiving a message from any of the other $n-1$ endpoints (excluding this) and forwarding it to any of the other $n-2$ endpoints (excluding this and the incoming).

Thus the maximum number of processes, p_{max}, for a given n can be found via the third degree polynomium:

$$p_{max}(n) = n^3 + n^2$$

If no security at all is required (neither integrity nor confidentiality), then one simply needs to create a process for the client sending directly and another sending via another node and for each endpoint create a process receiving directly, receiving via another node, and bouncing a message from one node to another. This summarizes the minimum number of processes as:

$$p_{min}(n) = 4n + 2$$

If policy requirements are mixed — and maybe even several policy requirements exist — then the number of processes will lie somewhere between these to extrema.

2.3.1.1 Security agreement

When having found two nodes, that need to communicate, then it is necessary to find their common integrity agreement, one can look at each of their agreements and see, which implementation is least possible, that will fulfill both their requirements.

If for instance one endpoint has a send policy requiring either secret-based or signature-based integrity must be used and another endpoint has a receive policy without integrity requirements, then the easiest implementation for the both of them is to use signature-based integrity (as this does not require a shared secret, that they must maintain and invalidate if leaked).

Some requirements are incompatible however. If one endpoint has a send policy requiring, that either secret-based or signature-based security is used — but not both (the `EXACTLYONE` combinator) — and this endpoint must communicate with another endpoint requiring that secret-based and signature-based both must be used (the `ALL` combinator), then they cannot agree on a common integrity policy and thus cannot communicate.

And incompatibility can be reached even simpler: if an endpoint has an integrity policy with a subject name claim, then only this subject name (the endpoint with that address) can communicate with it. And if the endpoint has several of these (joined under an `EXACTLYONE` combinator), then the set of subject names referenced can communicate with it — but not the ones not mentioned.

This creates a hierarchy of integrity policies with no requirements in the bottom and incompatible requirements in the top. Between these two is a lattice of security requirements. A security requirement in this lattice is then defined as belonging to the set R defined as:

$$R = \{p_{ALL}, p_{MORE}, p_{ONE} | \forall p \subseteq P\} \cup \{\epsilon, \xi\}$$

Where the set P is the set of all integrity policies.

ϵ indicates no requirement and ξ indicates incompatible requirements (the "ultimate", un-fulfillable, impossible requirement). The partial ordering of the elements of the set, $\sqsubseteq$, is defined as:

$$\forall x \in R: \quad \epsilon \sqsubseteq x \sqsubseteq \xi \tag{2.1}$$

$$\forall x, y \in R: \quad x \sqsubseteq y \quad \text{iff } x = p_{ALL} \wedge y = p'_{ALL} \wedge p \subseteq p' \tag{2.2}$$

$$\forall x, y \in R: \quad x \sqsubseteq y \quad \text{iff } x = p_{ONE} \wedge y = p'_{ONE} \wedge p' \subseteq p \tag{2.3}$$

$$\forall x, y \in R: \quad x \sqsubseteq y \quad \text{iff } x = p_{MORE} \wedge y = p'_{MORE} \wedge p' \subseteq p \tag{2.4}$$

$$\forall x, y \in R: \quad x \sqsubseteq y \quad \text{iff } x = p_{MORE} \wedge y = p'_{ALL} \wedge p' \subseteq p \tag{2.5}$$

$$\forall x, y \in R: \quad x \sqsubseteq y \quad \text{iff } x = p_{MORE} \wedge y = p'_{ONE} \wedge p' \subseteq p \tag{2.6}$$

That this is a partial order can be seen, as both reflexivity, antisymmetry and transitivity is valid for the set order $\subseteq$, and the above ordering merely wraps this order.

Rule 1 identify the top and bottom element of the order. Rules 2, 3 and 4 orders elements with the same combinator by the ordering of the sets of policies. It shows that for the `ALL` combinator, the fewer policies, the less restrictive, but for the `ONEORMORE` or `EXACTLYONE` combinators, the fewer policies, the more restrictive. The `ONEORMORE` combinator is compatible with both the other combinators as seen in rules 5 and 6, but they are both more restrictive.

Then we can define the *join* and *meet* operators, $\sqcup$ and $\sqcap$ respectively, by means of the partial ordering, $\sqsubseteq$, in the usual manner. For $x, y, z \in R$ we define:

$$z = x \sqcup y \quad \text{iff} \quad x \sqsubseteq z \wedge y \sqsubseteq z \wedge \forall z' \in R \text{ such that } x \sqsubseteq z' \wedge y \sqsubseteq z' : z \sqsubseteq z' \tag{2.7}$$

$$z = x \sqcap y \quad \text{iff} \quad z \sqsubseteq x \wedge z \sqsubseteq y \wedge \forall z' \in R \text{ such that } z' \sqsubseteq x \wedge z' \sqsubseteq y : z' \sqsubseteq z \tag{2.8}$$

Then we have a lattice, L, of requirements:

$$L = (R, \sqcup, \sqcap)$$

This definition can the be used to determine the set of integrity policy requirements between two processes for the implementation.

2.3.2 Creating processes

Earlier we defined seven sets of basic types of processes (in the worst-case scenario of process generation) and of these, 2 involves an endpoint and the client, 4 involves the client and two endpoints and the last involves three endpoints. Seen another way, 4 of them involves confidentiality policies (only client and ultimate receiver is involved in this), but all 7 involves integrity policies.

The way to generate all the necessary processes is to look at each of the fundamentally different types of processes, and for each examine all the combinations of nodes possible for this process type, and for each combination find the interfaces betwen the involved nodes, and for each interface determine the least fulfilling security agreement between the two — and then implement this. While doing this, if some interface has no security requirement, see if the resulting process might match another process completely (i.e. if another interface also has no security requirement), and if so, only include one of these in the final output.

For all security considerations it is implicitly expected, that the client has no security requirement of it's own — as no such requirement is possible to specify in the policy document.

In order to compare processes, we need a way to uniquely specify a process. A process in this context has the following properties:

- *name* — the name of the node, this process acts for. Will be either the client or one of the endpoints.
- *from* — the name of the node, that this process will receive a message from. Will be either the client, one of the endpoints, a generic node (if no identity is required) or empty (if this process does not receive a messge).
- *to* — the name of the node, that this process will send a message to. Will be either the client, one of the endpoints, a generic node (if no identity is required) or empty (if this process does not send a messge).
- *origin* — the originator of a message (only specified by the ultimate receiver), this can be the client, a node name or a generic node (if no confidentialy or integrity is required between the two).
- *destination* — the final destionation of a message (only specified by the initial sender), this can be the client, a node name or a generic node (if no confidentialy or integrity is required between the two).

These are defined in a tuple as:

$$\langle\langle m_{self}, m_{in}, m_{out}, m_{orig}, m_{dest} \rangle\rangle$$

Where these are defined as elements of:

$$\begin{aligned} m_{self} &\in M = \{\ client\ \} \cup \{\ node_i \mid \text{for } i \text{ between 1 and } n\ \} \\ m_{in}, m_{out}, m_{orig}, m_{dest} &\in M_0 = M \cup \{\ nil\ ,\ any\} \end{aligned}$$

Some invariants are imposed on top of this tuple structure regarding the node, that the process acts for:

- If the node is the message initiator (and thus specifies destination), the process does not receive a message.
- If the node is the ultimate message recipient (and thus specifies origin), the process does not send a message.
- If the node is the client, the node will either be the message initiator or the ultimate message recipient — and thus either specify origin or destination.

The processes will the be generated looking at each of the seven distinct types of processes, running through all the endpoints agreeing on security policies, and then if and when the individual agreements results in fixed recipients or senders use this as *from* or *to*, and if sender or recipient is not required to be a certain node, use *any* for the position in the tuple.

Chapter 3

Implementation

This chapter documents the execution of the theories behind the project. The execution is the software implementation necessary to create a program with the proper interface and architecture, that will enable end-users to utilize the verification, this project suggests.

The previous chapter contain several implementation ready sections (regarding parsing), but also sections merely theoretical, that require further development before being implementable. This especially concerns the outputting of LySatool input.

This chapter will first document the overall architecture of the solution. Which software components interact how, and what is necessary to implement to complete the tasks.

And furthermore, this chapter will also go into detail about the understanding og LySatool output in order to map this output (whether attacks are possible, and if so, where) back to the first the LySatool input, but then even further back to the policy document inputted by the user.

Finally this chapter will contain some of the key algorithms used in the implementation, that build the key parts of the project.

3.1 Architecture

An architectural overview is given in Figure 3.1.

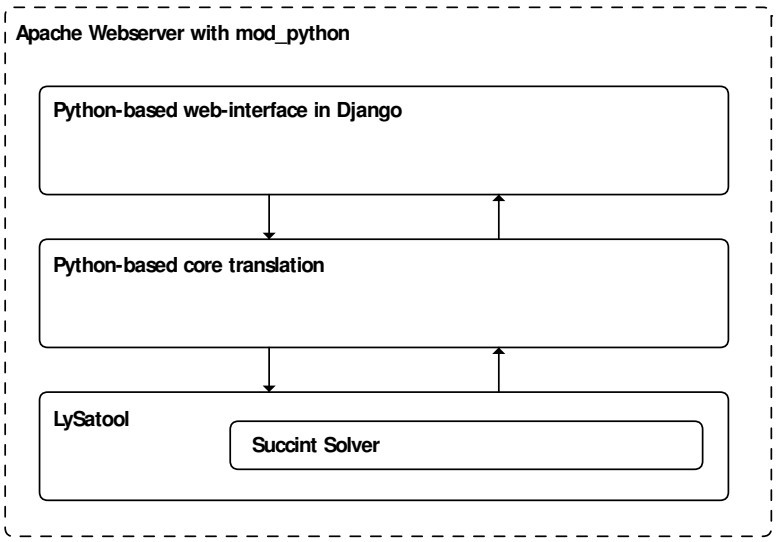

Figure 3.1: The whole application runs by invocation from the Apache Webserver. The interface and the engine is both implemented in Python using the LySatool as the verifier

The user interface will be HTTP-based, and the user will see a simple form for inputting or uploading the policy, and then the system will validate, that this is a real policy document conforming to the specification.

When inputted, the software will forward the document to the parser which will first establish the internal model, then generate LySatool input, then run this through LySa and interpret the results with respect to the internal representation, and finally comment on the results to the user with respect to the actual policy document inputted.

3.2 Pre-implementation decisions

As mentioned, not all theory is readily implementable, as some of it is written more abstract, than the Python language can comprehend. In order to ease the implementation, some further considerations about input and output for

LySatool will be given here.

3.2.1 Creating LySatool input

To utilize the methods developed in the previous chapter regarding process handling, a set of classes for processes as well as a join-method for integrity security requirement is necessary. The join-method is quite interesting, as it is formally defined in the previous chapter with respect to being the least of all upper bounds, but this is not implementable though it uniquely identifies the proper requirement — it simply does not specify how to find it.

In order to find the join of two security requirement as described in the previous chapter, an exhaustive matching is used matching all combinations of security requirements.

This uses the following procedure:

- If either of the two security requirements is ξ, return this, as this is the top element of the lattice.
- If either of the two security requirements is ϵ, return the other, as ϵ is the bottom element of the lattice.
- If the two security requirements are grouped under the same combinator:
  - If the combinator is `ONEORMORE` or `EXACTLYONE`, find the intersection of the two policy sets. If the set is empty, return ξ. Otherwise return the intersection under the same combinator as the two input requirements.
  - If the combinator is `ALL`, return the union of the two policy sets under the same combinator.
- If not the same combinator, then if either of the two security requirements is grouped under `ONEORMORE`:
  - If the policy set of the security requirement, that is grouped under `ONEORMORE` is a subset of the other requirements policy set, return the other requirement — as the first requirement is then fulfilled by the second.
  - If the above set is not a subset, return ξ as the requirements are incompatible.

- If neither is grouped under `ONEORMORE`, and they are not grouped under the same combinator, they must be grouped under `EXACTLYONE` and `ALL` respectively:
  - If the policy set grouped under `ALL` only contains one policy and this policy is a member of the other policy set (grouped under `EXACTLYONE`), return a policy set consisting of only this policy (and when only one policy, the combinator does not matter).
  - If any of the above two conditions are false, return ξ as the requirements are incompatible.

This can be seen implemented in Python like this:

```
STATUS_BOTTOM = 1
STATUS_MIDDLE = 2
STATUS_TOP    = 3
class ComparableIntegrityPolicySet:
  def __init__(self, status, combinator=None, policies=[]):
    self.status = status
    self.combinator = combinator
    self.policies = policies

  def join(a, b):
    # handle extremes
    if STATUS_TOP in [a.status, b.status]:
      return STATUS_TOP
    if a.status == STATUS_BOTTOM:
      return b
    if b.status == STATUS_BOTTOM:
      return a
    # handle middles
    if a.combinator == b.combinator:
      # same combinator - intersect or union
      policies = []
      if a.combinator in [COMBINATOR_MORE, COMBINATOR_ONE]:
        policies = a.policies.intersection(b.policies)
      else:
        policies = a.policies.union(b.policies)
      # if empty, incompatible - return TOP
      if len(policies) == 0:
        return ComparableIntegrityPolicySet(STATUS_TOP)
      # else return result
      return ComparableIntegrityPolicySet(STATUS_MIDDLE, COMBINATOR_MORE, policies)
    # now if one is MORE
    if b.combinator == COMBINATOR_MORE:
      # swap for ease of implementation - join is commutative anyway
      a, b = b, a
    if a.combinator == COMBINATOR_MORE:
      # if b's (ALL or ONE) set intersect a's (MORE) set, return b
      policies = a.policies.intersection(b.policies)
      if len(policies) > 0:
        return b
      # otherwise, impossible to join, return top
      return ComparableIntegrityPolicySet(STATUS_TOP)
    # one is ALL, the other is ONE
    if b.combinator == COMBINATOR_ONE:
      # swap for ease of implementation - join is commutative anyway
```

```
        a, b = b, a
      # only compatible if a (ALL) only has one entry, that is a member of b (ONE)
      if len(a.policies) == 1 and a.policies.issubset(b.policies):
        # then a is the join
        return a
50    # else impossible
      return ComparableIntegrityPolicySet(STATUS_TOP)
```

After this comes the actual generation of the LySa processes from the identified processes to be used.

This generation is done using some general simple rules for the protocol. When sending a message, the same amount of fields is always sent without concern whether all the fields are going to be used. This is done in order to have the intermediary nodes being able to forward the messages correctly without having to know, exactly which headers are being sent. The same number of headers are always expected — as well as the body.

The processes acting for the initial sender will be the only ones containing two-way encryption. And likewise, the processes acting for the ultimate receiver will be the only ones including two-way decryption. But all the processes acting for intermediary nodes will be validating and generating one-way encrypted values used for integrity enforcement.

Imagine for example a node, `n_1`, the acts as an intermediary node receiving a message from the client, with whom the node requires secret-based integrity, and forwarding this message to another node, `n_2`, with whom the node requires signature-based integrity (either `n_1` requires this on send or `n_2` requires this on receive). Then the resulting LySa process is:

```
(* wrap all processes in this shared key
   between the client and n_1 *)
(new Client_n_1_shared)(

(* receive message from client -
   headers for secret-based proxy are always
     From,
     To,
     Timestamp,
     IntegrityCheck
*)
(Client, N_1; timestamp).
({|Client, N_1, timestamp|}:Client_n_1_shared.
(* receive headers for main message -
```

```
   headers for message are always
     From,
     To,
     Action,
     PublicKey,
     Timestamp,
     IntegrityCheck
*)
(;mainfrom, mainto, mainaction, mainkey, maintimestamp).
(;maincheck).
(* receive message body *)
(;mainbody)
(* create public-private key pair
(new+- K).
(* send headers for signature-based proxy - always:
     From,
     To,
     PublicKey,
     Timestamp,
     IntegrityCheck
*)
(new localtimestamp)
<N_1, N_2, K+, localtimestamp>.
<{|localtimestamp|}:K->.
(* and then forward main message directly *)
<mainfrom, mainto, mainaction, mainkey, maintimestamp>.
<maincheck>.
<mainbody>.
0

)
```

This illustrates most of the procedures used to generate the LySatool input.

3.2.2 Understanding LySatool output

The output from the basic LySatool installation is currently an HTML-file. This file is directly displayable by a browser, but it is not easily understood, if the viewer does not fully understand the inner workings of LySa or the nature of the analysis.

Therefore, the important sections are extracted from this HTML output and then related back to first the internal representation created by the parser and from there, the results are related all the way back to the policy document.

This can be done, as the identifiers of the endpoints in the policy documents are kept all the way through to the LySatool input — and these identifiers are again extractable from the LySatool output and thus can the endpoint in question (for which some confidentiality is potentially broken) be identified.

The two most interesting parts of the LySatool output are by far:

- *Values that may not be confidential* — this part will highlight all publicly transferred values, that anyone will be able to intercept.
- *Violation of authentication properties* — this will highlight encrypted values, that might be decryptable by an attacker. It is only "might", as LySatool utilizes over-approximation, but any reported value should be of concern

3.3 Essential algorithms

Besides the previously shown algorithm for combining security requirements, other interesting aspects of the software includes for instance 7 scenarios identified for process generation.

As an example, take scenario 5. This was the scenario in which one of the endpoints acted as a node, that received a message from the client and proxied this forward to another endpoint. This process generation is done as simple as:

```
# scenario 5 - node from client to other node
for proxy in policyModel.stakeholders:
  for recipient in policyModel.stakeholders:
    if proxy == recipient:
      continue
    try:
      if uniqueRecipient(proxy, recipient):
        processes.add(model.Process(proxy.name,\
  fromm=model.NODE_CLIENT, to=recipient.name))
      else:
        processes.add(model.Process(proxy.name,\
```

```
fromm=model.NODE_CLIENT, to=model.NODE_ALL))
    except Uncombinable:
      continue
```

The key here is the `uniqueRecipient` method used to determine, whether or not the recipient of the two stakeholders must be uniquely identified as seen from the sender of the two. This is done by taking the request and response integrity policies respectively and matching them via the earlier shown join-method for security requirements.

If there is an incompatibility between the two nodes, an exception is thrown from within the `uniqueRecipient` method, and the whole process is skipped. This exception is thrown in one of two cases:

- If the join of the two requirements result in ξ — the top element — the two requirements are incompatible or
- If the recipient has a subjectname claim for his request policy, that requires, that only a certain subject is allowed to request this node, and the sender investigated is not this subject, then two requirements are likewise incompatible.

Chapter 4

Application

This chapter will demonstrate the state of the software at the completion of this project. The state is unfortunately not as advanced, as it could have been, but the essential aspects of the project will here be confirmed working.

The software is capable of understanding almost all variations of proper WSE policy documents and all the possibilities, that these offer. The software is also capable of translating these documents to a conceptually equivalent internal representation, that uses the same concepts of operations and policies of these.

The software is furthermore able to translate this concept of stakeholders into the concept of processes — where each process represent a real-life interface between one or more stakeholders interacting in the way, their policies require.

And finally, the software can export LySatool input equivalent of this notion of processes and input this to LySatool automatically for validation.

The software is unfortunately not at the current state able to interpret the results generated by LySatool automatically for user display. This seemed like a simple task, but unfortunately great difficulties arose in understanding the output with respect to the original input.

This chapter will run through two types of common usage a WSE policy doc-

uments and how this project's resulting software is able to understand and transform these.

4.1 Regular webservices

The first type of policy document is the simplest and yet the most common. It is simply a policy document describing a single endpoint with a single operation, and describing the request and response policy of this.

An example of such a document is:

```
<?xml version="1.0" encoding="utf-8"?>
<!--
 Description: Example file for testing simple remote service
 Author: Morten Barklund
-->
<policyDocument xmlns="http://schemas.microsoft.com/wse/2003/06/Policy">
  <mappings>
    <endpoint uri="http://www.example.invalid/service.ext">
      <operation requestAction="http://www.example.invalid/some_service.ext">
        <request policy="#request" />
        <response policy="#response" />
        <fault policy="" />
      </operation>
    </endpoint>
  </mappings>>
  <policies
    xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-
                wss-wssecurity-utility-1.0.xsd"
    xmlns:wsse="http://docs.oasis-open.org/wss/2004/01/oasis-200401-
                wss-wssecurity-secext-1.0.xsd"
    xmlns:wse="http://schemas.microsoft.com/wse/2003/06/Policy"
    xmlns:wsa="http://schemas.xmlsoap.org/ws/2004/03/addressing"
    xmlns:wssp="http://schemas.xmlsoap.org/ws/2002/12/secext"
    xmlns:wsp="http://schemas.xmlsoap.org/ws/2002/12/policy"
    xmlns:wssc="http://schemas.xmlsoap.org/ws/2004/04/sc"
    xmlns:rp="http://schemas.xmlsoap.org/rp"
  >
    <wsp:Policy wsu:Id="request">
      <ALL>
        <ONEORMORE>
          <wssp:Integrity>
            <wssp:TokenInfo>
              <wssp:SecurityToken>
                <wssp:TokenType>
                  http://docs.oasis-open.org/wss/2004/01/oasis-200401-
                    wss-username-token-profile-1.0#UsernameToken
                </wssp:TokenType>
                <wssp:Claims>
                  <wse:SubjectName>http://www.proxy.invalid/proxy.ext</wse:SubjectName>
                </wssp:Claims>
              </wssp:SecurityToken>
            </wssp:TokenInfo>
            <wssp:MessageParts
              xmlns:rp="http://schemas.xmlsoap.org/rp"
```

```
            Dialect="http://schemas.xmlsoap.org/2002/12/wsse#part"
          >
            wsp:Body() wse:Timestamp() wse:Header(wsp:To)
          </wssp:MessageParts>
        </wssp:Integrity>
        <wssp:Integrity>
          <wssp:TokenInfo>
            <wssp:SecurityToken>
              <wssp:TokenType>
                http://docs.oasis-open.org/wss/2004/01/oasis-200401-
                  wss-x509-token-profile-1.0#X509v3
              </wssp:TokenType>
              <wssp:Claims>
                <wse:SubjectName>http://www.proxy.invalid/proxy.ext</wse:SubjectName>
              </wssp:Claims>
            </wssp:SecurityToken>
          </wssp:TokenInfo>
          <wssp:MessageParts
            xmlns:rp="http://schemas.xmlsoap.org/rp"
            Dialect="http://schemas.xmlsoap.org/2002/12/wsse#part"
          >
            wsp:Body() wse:Timestamp() wse:Header(wsp:To)
          </wssp:MessageParts>
        </wssp:Integrity>
      </ONEORMORE>
      <EXACTLYONE>
        <wssp:Confidentiality>
          <wssp:TokenInfo>
            <wssp:SecurityToken>
              <wssp:TokenType>
                http://docs.oasis-open.org/wss/2004/01/oasis-200401-
                  wss-username-token-profile-1.0#UsernameToken
              </wssp:TokenType>
            </wssp:SecurityToken>
          </wssp:TokenInfo>
          <wssp:MessageParts
            xmlns:rp="http://schemas.xmlsoap.org/rp"
            Dialect="http://schemas.xmlsoap.org/2002/12/wsse#part"
          >
            wsp:Body()
          </wssp:MessageParts>
        </wssp:Confidentiality>
        <wssp:Confidentiality>
          <wssp:TokenInfo>
            <wssp:SecurityToken>
              <wssp:TokenType>
                http://docs.oasis-open.org/wss/2004/01/oasis-200401-
                  wss-x509-token-profile-1.0#X509v3
              </wssp:TokenType>
            </wssp:SecurityToken>
          </wssp:TokenInfo>
          <wssp:MessageParts
            xmlns:rp="http://schemas.xmlsoap.org/rp"
            Dialect="http://schemas.xmlsoap.org/2002/12/wsse#part"
          >
            wsp:Body()
          </wssp:MessageParts>
        </wssp:Confidentiality>
      </EXACTLYONE>
    </ALL>
  </wsp:Policy>
  <wsp:Policy wsu:Id="response">
    <ALL>
      <wssp:Integrity>
        <wssp:TokenInfo>
```

```
            <wssp:SecurityToken>
                <wssp:TokenType>
                  http://docs.oasis-open.org/wss/2004/01/oasis-200401-
                    wss-x509-token-profile-1.0#X509v3
                </wssp:TokenType>
              </wssp:SecurityToken>
            </wssp:TokenInfo>
            <wssp:MessageParts
              xmlns:rp="http://schemas.xmlsoap.org/rp"
                Dialect="http://schemas.xmlsoap.org/2002/12/wsse#part"
            >
              wsp:Body() wse:Timestamp() wse:Header(wsp:To)
            </wssp:MessageParts>
          </wssp:Integrity>
          <EXACTLYONE>
            <wssp:Confidentiality>
              <wssp:TokenInfo>
                <wssp:SecurityToken>
                  <wssp:TokenType>
                    http://docs.oasis-open.org/wss/2004/01/oasis-200401-
                      wss-username-token-profile-1.0#UsernameToken
                  </wssp:TokenType>
                </wssp:SecurityToken>
              </wssp:TokenInfo>
              <wssp:MessageParts
                xmlns:rp="http://schemas.xmlsoap.org/rp"
                Dialect="http://schemas.xmlsoap.org/2002/12/wsse#part"
              >
                wsp:Body()
              </wssp:MessageParts>
            </wssp:Confidentiality>
            <wssp:Confidentiality>
              <wssp:TokenInfo>
                <wssp:SecurityToken>
                  <wssp:TokenType>
                    http://docs.oasis-open.org/wss/2004/01/oasis-200401-
                      wss-x509-token-profile-1.0#X509v3
                  </wssp:TokenType>
                </wssp:SecurityToken>
              </wssp:TokenInfo>
              <wssp:MessageParts
                xmlns:rp="http://schemas.xmlsoap.org/rp"
                Dialect="http://schemas.xmlsoap.org/2002/12/wsse#part"
              >
                wsp:Body()
              </wssp:MessageParts>
            </wssp:Confidentiality>
          </EXACTLYONE>
        </ALL>
      </wsp:Policy>
    </policies>
  </policyDocument>
```

This document specifies many security policies and in particular several integrity policies both regarding requests and responses, as well as two different encryption policies for confidential requests — but on the other hand only a single possible encryption policy for responses.

To convert this single stakeholder (one endpoint, one operation) to a notion of processes, it is simply a matter of creating one process for the client, that is to

interact with this stakeholder, and another process representing the stakeholder.

4.2 SOAP Message Chain validation

Simple single-stakeholder policy documents are however not very interesting — as they are very easy to comprehend by the developer, and as the number of processes is very small, namely 2.

More interesting and more true to the nature of SOAP is an example of a policy document describing several stakeholders, and somewhat linking them in a SOAP message chain.

A rather exotic two-stakeholder example, that has been used as a reference for the implementation, is the following document:

```
<?xml version="1.0" encoding="utf-8"?>
<!--
 Description: Example file for testing proxy access to remote service
 Author: Morten Barklund
-->
<policyDocument xmlns="http://schemas.microsoft.com/wse/2003/06/Policy">
  <mappings>
    <endpoint uri="http://www.example.invalid/service.ext">
      <operation requestAction="http://www.example.invalid/some_service.ext">
        <request policy="#service_request" />
        <response policy="#service_response" />
        <fault policy="" />
      </operation>
    </endpoint>
    <endpoint uri="http://www.proxy.invalid/proxy.ext">
      <defaultOperation>
        <request policy="#proxy_request" />
        <response policy="#proxy_response" />
        <fault policy="" />
      </defaultOperation>
    </endpoint>
  </mappings>
  <policies
    xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-
                wss-wssecurity-utility-1.0.xsd"
    xmlns:wsse="http://docs.oasis-open.org/wss/2004/01/oasis-200401-
                 wss-wssecurity-secext-1.0.xsd"
    xmlns:wse="http://schemas.microsoft.com/wse/2003/06/Policy"
    xmlns:wsa="http://schemas.xmlsoap.org/ws/2004/03/addressing"
    xmlns:wssp="http://schemas.xmlsoap.org/ws/2002/12/secext"
    xmlns:wsp="http://schemas.xmlsoap.org/ws/2002/12/policy"
    xmlns:wssc="http://schemas.xmlsoap.org/ws/2004/04/sc"
    xmlns:rp="http://schemas.xmlsoap.org/rp"
  >
    <wsp:Policy wsu:Id="service_request">
      <EXACTLYONE>
        <wssp:Integrity>
          <wssp:TokenInfo>
```

```
            <wssp:SecurityToken>
              <wssp:TokenType>
                http://docs.oasis-open.org/wss/2004/01/oasis-200401-
                  wss-username-token-profile-1.0#UsernameToken
              </wssp:TokenType>
              <wssp:Claims>
                <wse:SubjectName>
                  http://www.example.invalid/proxy.ext
                </wse:SubjectName>
              </wssp:Claims>
            </wssp:SecurityToken>
          </wssp:TokenInfo>
          <wssp:MessageParts
           xmlns:rp="http://schemas.xmlsoap.org/rp"
           Dialect="http://schemas.xmlsoap.org/2002/12/wsse#part">
            wsp:Body() wse:Timestamp() wse:Header(wsp:To)
          </wssp:MessageParts>
        </wssp:Integrity>
        <wssp:Integrity>
          <wssp:TokenInfo>
            <wssp:SecurityToken>
              <wssp:TokenType>
                http://docs.oasis-open.org/wss/2004/01/oasis-200401-
                  wss-x509-token-profile-1.0#X509v3
              </wssp:TokenType>
              <wssp:Claims>
                <wse:SubjectName>
                  http://www.example.invalid/proxy.ext
                </wse:SubjectName>
              </wssp:Claims>
            </wssp:SecurityToken>
          </wssp:TokenInfo>
          <wssp:MessageParts
           xmlns:rp="http://schemas.xmlsoap.org/rp"
           Dialect="http://schemas.xmlsoap.org/2002/12/wsse#part">
            wsp:Body() wse:Timestamp() wse:Header(wsp:To)
          </wssp:MessageParts>
        </wssp:Integrity>
      </EXACTLYONE>
    </wsp:Policy>
    <wsp:Policy wsu:Id="service_response">
      <ALL>
        <wssp:Integrity>
          <wssp:TokenInfo>
            <wssp:SecurityToken>
              <wssp:TokenType>
                http://docs.oasis-open.org/wss/2004/01/oasis-200401-
                  wss-x509-token-profile-1.0#X509v3
              </wssp:TokenType>
            </wssp:SecurityToken>
          </wssp:TokenInfo>
          <wssp:MessageParts
           xmlns:rp="http://schemas.xmlsoap.org/rp"
           Dialect="http://schemas.xmlsoap.org/2002/12/wsse#part">
            wsp:Body() wse:Timestamp() wse:Header(wsp:To)
          </wssp:MessageParts>
        </wssp:Integrity>
        <wssp:Confidentiality>
          <wssp:KeyInfo>
            <wssp:SecurityToken>
              <wssp:TokenType>
                http://docs.oasis-open.org/wss/2004/01/oasis-200401-
                  wss-x509-token-profile-1.0#X509v3
              </wssp:TokenType>
            </wssp:SecurityToken>
```

```
            </wssp:KeyInfo>
            <wssp:MessageParts
              xmlns:rp="http://schemas.xmlsoap.org/rp"
              Dialect="http://schemas.xmlsoap.org/2002/12/wsse#part">
                wsp:Body()
            </wssp:MessageParts>
          </wssp:Confidentiality>
        </ALL>
        </wsp:Policy>
        <wsp:Policy wsu:Id="proxy_request">
          <wssp:Integrity>
            <wssp:TokenInfo>
              <wssp:SecurityToken>
                <wssp:TokenType>
                  http://docs.oasis-open.org/wss/2004/01/oasis-200401-
                    wss-username-token-profile-1.0#UsernameToken
                </wssp:TokenType>
              </wssp:SecurityToken>
            </wssp:TokenInfo>
              <wssp:MessageParts
               xmlns:rp="http://schemas.xmlsoap.org/rp"
               Dialect="http://schemas.xmlsoap.org/2002/12/wsse#part">
                 wsp:Body() wse:Timestamp() wse:Header(wsp:To)
              </wssp:MessageParts>
          </wssp:Integrity>
        </wsp:Policy>
        <wsp:Policy wsu:Id="proxy_response">
          <wssp:Integrity>
            <wssp:TokenInfo>
              <wssp:SecurityToken>
                <wssp:TokenType>
                  http://docs.oasis-open.org/wss/2004/01/oasis-200401-
                    wss-username-token-profile-1.0#UsernameToken
                </wssp:TokenType>
              </wssp:SecurityToken>
            </wssp:TokenInfo>
              <wssp:MessageParts
               xmlns:rp="http://schemas.xmlsoap.org/rp"
               Dialect="http://schemas.xmlsoap.org/2002/12/wsse#part">
                       wsp:Body() wse:Timestamp() wse:Header(wsp:To)
              </wssp:MessageParts>
          </wssp:Integrity>
        </wsp:Policy>
      </policies>
    </policyDocument>
```

This document defines two stakeholders:

- A remote service, that is the interesting ultimate receiver for the clients to use and
- A proxy service, which is the only service, that is allowed to call the remote service (due to the subject name claim on the request policies of the remote service).

This could for instance easily display how a remote service is to be used from inside a company. The service is situated outside the company network, but

only a trusted node on the company network is allowed to access this external service. This single trusted node and the external service share a key, that allows them to communicate via secret-based integrity and confidentiality.

Then any client inside the company can send a request to the remote service via the internal company proxy node and in the process send integrity headers needed to authorize with the internal company node as well as the headers needed to authorize with the external node.

This setup generates 8 processes, that combine all the ways the individual nodes can communicate. Note that this includes, that the client talks directly to the proxy, and the proxy replies directly to the client. These two processes would in practice not be interesting, as the internal proxy node cannot do anything itself, but if targeted, it must reply (maybe just a fault), as that is the requirement for all SOAP nodes.

Chapter 5

Conclusion

Upon completion of this project, several items are still open. First of all, there is the value of the verification of SOAP policy documents completed through this project. Is it good, is it interesting? To these questions, the short answer is "yes", but a longer evaluation will follow later.

Besides this, the is the key elements used as a foundation for this project. What is their status, what is their future, where does this project fit in? This will result in a short evaluation of the whole SOAP technology and discussion of where SOAP is heading if anywhere. There will also be a discussion of the possibilities of Lysatool with respect to Webservices and the modern web standards for communication. Finally, the status of Tulafale and it's result compared to this project will be discussed.

5.1 SOAP protocol validation

The key part of this project is to establish a foundation for SOAP protocol validation via LySatool — and I strongly believe, that it accomplishes just this. The result is not final, it lacks a complete working prototype, but it lays a solid foundation to continue development on top of.

Personally I will of course continue the development of this in my spare time, as this subject is not only relevant to both the academic and professional communities, but certainly also relevant for me personally, as web standards and protocol definitions is a huge part of my job. I will thus keep regular updates to this research project on my community site, where the current version of the program also can be seen as seen as the final part of the conversion of LySatool output has been completed within a short time.

This site can be seen here:

http://barklund.org

The main parts to extend, when continueing the work on this project, is of course to complete the user experience circle and let users get the full benefit of the current small implementation. Later, extensions to the language and implementation can be considered — for instance extended to include WSDL files and maybe even compare WSE policy documents to WSDL web service specifications. This would definitely be a very interesting way to go with WSDL being the (most likely) future specification for web service interoperability.

Other improvements for the software would be a refactorization. The parsing and outputting could be done more smoothly and with better utilization of external libraries for ease of implementation. This has however not been a concern — the primary purpose was to build a working, complete parsing and translation.

5.2 Web services

As mentioned SOAP and WSDL are among the standards, that are leading the field. But other standards for interoperability across platforms, programming languages and programming environments and frameworks exist. XML-RPC, JSON, AMF just to name a few. Some arose for just this purpose — like SOAP or XML-RPC — others initially were specified for other purposes — like JSON for JavaScript and AMF for Flash Media Server — but has since proven highly interoperable and easy to implement in many other contexts.

Many standards are being created and used by the open source community, but the major standards still start at the major companies. SOAP started with Microsoft, AMF started at Adobe, and XML-RPC is actually a predecessor for

SOAP, but much simpler and still in use in its simple form — and thus also came from research projects at Microsoft.

And WSDL isn't the only standard trying to organize SOAP protocols. Another much more formal specification language is the Web Service Conversation Language 1.0[1] originally suggested by Hewlett-Packard to W3C in 2002. This language defines web services and their handling of input and output via finite state automatons.

Other standards go in the direction of web service registries, where for instance Web Service Inspection Language[15] is a method for reflecting current web services in a manner suitable for inclusion in larger registries. This project is a joint-venture between Microsoft and IBM.

If concentrating of the web of today, the primary single language used for web service interoperability is clearly SOAP.

But that does not mean, that SOAP has a large market share — it just means, that there a so many different specifications and standards, and even more just creating there own as they go along. This is an increasing problem urging the need for specifications pleasing everyone. And SOAP is a likely candidate.

5.3 LySaTool

This project documents a different usage of LySatool — namely as an engine behind another engine for doing the complete modelling of processes and attackers, but without any understanding of who communicates with whom and via whom.

LySatool is used as an environment in which all possible message paths can be run simultaneously in a complex setup of many different services across many systems and networks.

And more importantly, this illustrate a usage of LySa, that does not require knowledge of the quite unhandy — but very formal — input language, that LySatool expects, nor does it require any knowledge to the complexity of the analysises, that LySa performs. The only thing that matters to the end user is if and where, attacks might occur.

5.3.1 Improvements

But the conversion of policy documents to LySatool input is very complex and requires many processes to describe even simple policy mappings with few endpoints.

The main improvements of LySa, that this project leaves me with, are improvements, that I cannot fully see the consequences of with respect to the current workings of LySatool.

Improvements such as structures (terms of terms) would seem doable, as one could actually see encryption and decryption using a public "secret" as a method for sending structures inside single terms. Thus such an extension seem quite straight-forward, but other issues might occur, when going in-depth with the proposal.

Other suggestions include making the LySatool input language for *feature rich* including syntactic "sugar" for creating loops, conditions and similar features in order to write more complex processes the can communicate natively in the complex ways, that web service stakeholders might do.

Such suggestions are much more fundamental, and will require much more care and consideration before attempted included in LySa.

5.4 Tulafale

Tulafale was originally the inspiration for this project, but when examining, what it really was, it turned out to be so closely coupled with another software, that the stand-alone version, that Microsoft Research released, was flawed and uninteresting, as it did not even accept the proper format as well as only supported a minor subset of the specification.

The conclusion on this project with respect to Tulafale is most definitely that this research project might be interesting for Microsoft themselves in an attempt to build better software themselves, but to the community, the project is a bit irrelevant — and I have not found anyone actually using the stand-alone tool Tulafale for anything serious.

Appendix A

Program source

This appendix includes source files. This is both Python source files as well as XML source files.

A.1 Python source

The Python source files are the main source files for building and utilizing the model necessary to convert between the two forms.

A.1.1 `model.py`

```
class Model:
  def __init__(self, stakeholders=[]):
    self.stakeholders = stakeholders


COMBINATOR_ALL  = 1
COMBINATOR_MORE = 2
COMBINATOR_ONE  = 3

class Message:
  def __init__(self, headers=[], body=False):
```

```
    self.headers = headers
    self.body = body

class ConfidentialityPolicySet:
  def __init__(self, policies=set(), combinator=COMBINATOR_ONE):
    self.combinator = combinator
    self.policies = policies

class IntegrityPolicySet:
  def __init__(self, policies=set(), combinator=COMBINATOR_MORE):
    self.policies = policies
    self.combinator = combinator

class ConfidentialityPolicy:
  def __init__(self, token=None, claims=[]):
    self.token = token
    self.claims = claims

  def __hash__(self):
    return hash(self.token)

  def __eq__(self, other):
    return self.token == other.token

class Policy:
  def __init__(self, message=Message(), intPolicies=IntegrityPolicySet(), confPolicies=ConfidentialityPolicySet()):
    self.message = message
    self.confidentiality = confPolicies
    self.integrity = intPolicies

class Stakeholder:
  def __init__(self, name, request=Policy(), response=Policy()):
    self.name = name
    self.request = request
    self.response = response

TOKENVALUE_X509     = 'http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-x509-token-profile-1.0#X509v3'
TOKENVALUE_USERNAME = 'http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-username-token-profile-1.0#UsernameTo
TOKEN_SECRET        = 1
TOKEN_PUBLIC        = 2
TOKEN_SIGNATURE     = 3
class TokenType:
  def __init__(self, tokenType=None, key=None):
    self.tokenType = tokenType
    self.key = key

  def __hash__(self):
    return self.tokenType

CLAIM_SUBJECT = 1
CLAIM_ROLE    = 2
class Claim:
  def __init__(self, claimType, value):
    self.claimType = claimType
    self.value = value

class IntegrityPolicy:
  def __init__(self, token=None, claims=[], parts=set()):
    self.token = token
    self.claims = claims
    self.parts = parts

  def __hash__(self):
    # create hash from first 4 bytes of md5 hash
    b = md5.new('$ %s $ %s $' % (self.token, ",".join(self.parts))).digest()
```

```
    return (ord(b[0])) | (ord(b[1])<<8) | (ord(b[2])<<16) | (ord(b[3])<<23)

  def __eq__(self, other):
    b1 = self.token == other.token
    b2 = len(self.parts) == len(self.parts.intersection(other.parts))
    return b1 and b2


PART_BODY   = 1
PART_HEADER = 2
class MessagePart:
  def __init__(self, part, value):
    self.part = part
    self.value = value

# for generating output - immutable, comparable class
import md5
NODE_CLIENT  = -1
NODE_ALL     = -2
NODE_NONE    = -3
NODE_UNKNOWN = -4
class Process:
  def __init__(self, name, fromm=NODE_NONE, to=NODE_NONE, origin=NODE_UNKNOWN, destination=NODE_UNKNOWN):
    self.name = name
    self.fromm = fromm
    self.to = to
    self.origin = origin
    self.destination = destination

  def __hash__(self):
    # create hash from first 4 bytes of md5 hash
    b = md5.new('$ %s $ %s $ %s $ %s $ %s $ %s $' % (self.name, self.fromm, self.to, self.origin, self.destination))
    return (ord(b[0])) | (ord(b[1])<<8) | (ord(b[2])<<16) | (ord(b[3])<<23)

  def __eq__(self, other):
    b1 = self.name == other.name
    b2 = self.fromm == other.fromm
    b3 = self.to == other.to
    b4 = self.origin == other.origin
    b5 = self.destination == other.destination
    return b1 and b2 and b3 and b4 and b5

STATUS_BOTTOM = 1
STATUS_MIDDLE = 2
STATUS_TOP    = 3
class ComparableIntegrityPolicySet:
  def __init__(self, status, combinator=None, policies=[]):
    self.status = status
    self.combinator = combinator
    self.policies = policies

  def join(a, b):
    # handle extremes
    if STATUS_TOP in [a.status, b.status]:
      return STATUS_TOP
    if a.status == STATUS_BOTTOM:
      return b
    if b.status == STATUS_BOTTOM:
      return a
    # handle middles
    if a.combinator == b.combinator:
      # same combinator - intersect or union
      policies = []
      if a.combinator in [COMBINATOR_MORE, COMBINATOR_ONE]:
        policies = a.policies.intersection(b.policies)
```

```
      else:
        policies = a.policies.union(b.policies)
      # if empty, incompatible - return TOP
      if len(policies) == 0:
        return ComparableIntegrityPolicySet(STATUS_TOP)
      # else return result
      return ComparableIntegrityPolicySet(STATUS_MIDDLE, COMBINATOR_MORE, policies)
    # now if one is MORE
    if b.combinator == COMBINATOR_MORE:
      # swap for ease of implementation - join is commutative anyway
      a, b = b, a
    if a.combinator == COMBINATOR_MORE:
      # if b's (ALL or ONE) set intersect a's (MORE) set, return b
      policies = a.policies.intersection(b.policies)
      if len(policies) > 0:
        return b
      # otherwise, impossible to join, return top
      return ComparableIntegrityPolicySet(STATUS_TOP)
    # one is ALL, the other is ONE
    if b.combinator == COMBINATOR_ONE:
      # swap for ease of implementation - join is commutative anyway
      a, b = b, a
    # only compatible if a (ALL) only has one entry, that is a member of b (ONE)
    if len(a.policies) == 1 and a.policies.issubset(b.policies):
      # then a is the join
      return a
    # else impossible
    return ComparableIntegrityPolicySet(STATUS_TOP)
```

A.1.2 readers.py

```
from string import lower
import model

def translateDocument(node):
  # get dict of policies
  policyMap = translatePolicies(getFirstChildByName(node, 'policies'))

  # find endpoints and prepare to create stakeholders
  mappings = getFirstChildByName(node, 'mappings')
  if mappings == None:
    return None
  endpoints = list(mappings.childNodes)
  stakeholders = []
  for endpoint in endpoints:
    operations = list(endpoint.childNodes)
    for operation in operations:
      # for each operation, find policies and name to create stakeholder
      request = model.Policy()
      response = model.Policy()
      try:
        requestId = getFirstChildByName(operation, 'request').getAttribute('policy').strip("# ")
        request = policyMap[requestId]
        responseId = getFirstChildByName(operation, 'response').getAttribute('policy').strip("# ")
        response = policyMap[responseId]
      except:
        pass
      name = None
      if lower(operation.localName) == 'defaultoperation':
        name = endpoint.getAttribute('uri')
      else:
```

```
            name = operation.getAttribute('requestAction')
          stakeholders.append(model.Stakeholder(name, request, response))

    return model.Model(stakeholders)

  def translatePolicies(policiesNode):
    policyMap = {}
    for policyNode in list(policiesNode.childNodes):
      headers = extractHeaders(policyNode)
      body = extractBody(policyNode)
      message = model.Message(headers, body)
      integrity = extractIntegritySet(policyNode)
      confidentiality = extractConfidentialitySet(policyNode)
      policy = model.Policy(message, integrity, confidentiality)
      policyId = policyNode.getAttribute(("http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-uti
      policyMap[policyId] = policy
    return policyMap

  def extractHeaders(policyNode):
    headers = []
    for part in getChildNodesByName(policyNode, "messageparts"):
      headers += part.firstChild.nodeValue.split(" ")
    return headers

  def extractBody(policyNode):
    for part in getChildNodesByName(policyNode, "messageparts"):
      if 'body' in part.firstChild.nodeValue.split(" "):
        return True
    return False

  def extractConfidentialitySet(policyNode):
    combinator = model.COMBINATOR_ONE
    policySet = set()
    policies = getChildNodesByName(policyNode, "confidentiality")
    # special handling of zero or one
    if len(policies) == 0:
      return model.ConfidentialityPolicySet(policySet)
    if len(policies) == 1:
      policySet.add(translateConfidentiality(policies[0]))
      return model.ConfidentialityPolicySet(policySet)
    # more than one, check combinator
    if lower(policies[0].parentNode.localName) != "exactlyone":
      # generate warning
      pass
    for policy in policies:
      policySet.add(translateConfidentiality(policy))
    return model.ConfidentialityPolicySet(policySet, combinator)

  def translateConfidentiality(confidentiality):
    # first, extract token type
    tokenType = None
    tokenTypeNode = getFirstChildByName(confidentiality, "tokentype")
    if tokenTypeNode != None:
      tokenValue = tokenTypeNode.firstChild.nodeValue.strip()
      if tokenValue == model.TOKENVALUE_X509:
        reference = getFirstChildByName(confidentiality, "securitytokenreference")
        if reference != None:
          tokenType = model.TokenType(model.TOKEN_PUBLIC, reference.firstChild.nodeValue)
        else:
          tokenType = model.TokenType(model.TOKEN_PUBLIC)
      elif tokenValue == model.TOKENVALUE_USERNAME:
        tokenType = model.TokenType(model.TOKEN_SECRET)
    # get claims
    claims = []
    subjectName = getFirstChildByName(confidentiality, "subjectname")
```

```
    if subjectName != None:
      claims.append(model.Claim(model.CLAIM_SUBJECT, subjectName.firstChild.nodeValue))
    role = getFirstChildByName(confidentiality, "role")
    if role != None:
      claims.append(model.Claim(model.CLAIM_ROLE, role.firstChild.nodeValue))
    # return policy
    return model.ConfidentialityPolicy(tokenType, claims)


def extractIntegritySet(policyNode):
    combinator = None
    policySet = set()
    policies = getChildNodesByName(policyNode, "integrity")
    # special handling of zero or one
    if len(policies) == 0:
      return model.IntegrityPolicySet(policySet)
    if len(policies) == 1:
      policySet.add(translateIntegrity(policies[0]))
      return model.IntegrityPolicySet(policySet)
    # more than one, check combinator
    if lower(policies[0].parentNode.localName) != "exactlyone":
      combinator = model.COMBINATOR_ONE
    elif lower(policies[0].parentNode.localName) != "all":
      combinator = model.COMBINATOR_ALL
    elif lower(policies[0].parentNode.localName) != "oneormore":
      combinator = model.COMBINATOR_MORE
    for policy in policies:
      policySet.add(translateIntegrity(policy))
    return model.IntegrityPolicySet(policySet, combinator)

def translateIntegrity(integrity):
    # first, extract token type
    tokenType = None
    tokenTypeNode = getFirstChildByName(integrity, "tokentype")
    if tokenTypeNode != None:
      tokenValue = tokenTypeNode.firstChild.nodeValue
      if tokenValue == model.TOKENVALUE_X509:
        tokenType = model.TOKEN_SIGNATURE
      elif tokenValue == model.TOKENVALUE_USERNAME:
        tokenType = model.TOKEN_SECRET
    # get claims
    claims = []
    subjectName = getFirstChildByName(integrity, "subjectname")
    if subjectName != None:
      claims.append(model.Claim(model.CLAIM_SUBJECT, subjectName.firstChild.nodeValue))
    role = getFirstChildByName(integrity, "role")
    if role != None:
      claims.append(model.Claim(model.CLAIM_ROLE, role.firstChild.nodeValue))
    # get messageparts
    parts = set()
    partsNode = getFirstChildByName(integrity, "messageparts")
    if partsNode != None:
      for part in partsNode.firstChild.nodeValue.split(" "):
        parts.add(part)
    # return policy
    return model.IntegrityPolicy(tokenType, claims, parts)




# helper functions below

# get node by name - recursive
def getFirstChildByName(node, name):
```

```
    # check direct
    for child in list(node.childNodes):
      if child.localName == None:
        continue
      if lower(child.localName) == lower(name):
        return child
    # check children
    for child in list(node.childNodes):
      n = getFirstChildByName(child, name)
      if n != None:
        return n
    # too bad
    return None

# get all nodes by name - recursive
def getChildNodesByName(node, name):
  nodes = []
  for child in list(node.childNodes):
    if child.localName == None:
      continue
    if lower(child.localName) == lower(name):
      # don't recurse when found
      nodes.append(child)
    else:
      nodes += getChildNodesByName(child, name)
  return nodes
```

A.1.3 writers.py

```
import model
import sys, traceback

# generally, the client has no policies
client = model.Stakeholder("client")

# generic stakeholder is sometimes needed
generic = model.Stakeholder("generic")

def getLySaProcesses(policyModel):
  processes = set()

  # scenario 1 - client direct to endpoint
  for stakeholder in policyModel.stakeholders:
    try:
      if uniqueRecipientFull(client, stakeholder):
        processes.add(model.Process(model.NODE_CLIENT, to=stakeholder.name, destination=stakeholder.name))
      else:
        processes.add(model.Process(model.NODE_CLIENT, to=model.NODE_ALL, destination=model.NODE_ALL))
    except Uncombinable:
      continue

  # scenario 2 - client to endpoint via node
  for ultimate in policyModel.stakeholders:
    destination = model.NODE_ALL
    try:
      if uniqueRecipientFull(client, ultimate):
        destination = ultimate.name
    except Uncombinable:
      pass
    for proxy in policyModel.stakeholders:
      if proxy == ultimate:
```

```
        continue
      try:
        if uniqueRecipient(client, proxy):
          processes.add(model.Process(model.NODE_CLIENT, to=proxy.name, destination=destination))
        else:
          processes.add(model.Process(model.NODE_CLIENT, to=model.NODE_ALL, destination=destination))
      except Uncombinable:
        continue

# scenario 3 - ultimate receiver direct from client - all are unique
for stakeholder in policyModel.stakeholders:
  processes.add(model.Process(stakeholder.name, fromm=model.NODE_CLIENT, origin=model.NODE_CLIENT))

# scenario 4 - ultimate receiver from client via node
for ultimate in policyModel.stakeholders:
  for proxy in policyModel.stakeholders:
    if proxy == ultimate:
      continue
    try:
      if uniqueSender(proxy, ultimate):
        processes.add(model.Process(ultimate.name, fromm=proxy.name, origin=model.NODE_CLIENT))
      else:
        processes.add(model.Process(ultimate.name, fromm=model.NODE_ALL, origin=model.NODE_CLIENT))
    except Uncombinable:
      continue

# scenario 5 - node from client to other node
for proxy in policyModel.stakeholders:
  for recipient in policyModel.stakeholders:
    if proxy == recipient:
      continue
    try:
      if uniqueRecipient(proxy, recipient):
        processes.add(model.Process(proxy.name, fromm=model.NODE_CLIENT, to=recipient.name))
      else:
        processes.add(model.Process(proxy.name, fromm=model.NODE_CLIENT, to=model.NODE_ALL))
    except Uncombinable:
      continue

# scenario 6 - node from other node to client
for proxy in policyModel.stakeholders:
  for sender in policyModel.stakeholders:
    if proxy == sender:
      continue
    try:
      if proxy == sender:
        continue
      if uniqueSender(sender, proxy):
        processes.add(model.Process(proxy.name, to=model.NODE_CLIENT, fromm=sender.name))
      else:
        processes.add(model.Process(proxy.name, to=model.NODE_CLIENT, fromm=model.NODE_ALL))
    except Uncombinable:
      continue

# scenario 7 - node from other node to third node
for proxy in policyModel.stakeholders:
  for sender in policyModel.stakeholders:
    if proxy == sender:
      continue
    fromm = model.NODE_ALL
    try:
      if uniqueSender(sender, proxy):
        fromm = sender.name
    except Uncombinable:
      pass
```

```
      for recipient in policyModel.stakeholders:
        if recipient in [proxy, sender]:
          continue
        try:
          if uniqueRecipient(proxy, sender):
            processes.add(model.Process(proxy.name, fromm=fromm, to=recipient.name))
          else:
            processes.add(model.Process(proxy.name, fromm=fromm, to=model.NODE_ALL))
        except Uncombinable:
          continue

  return processes

def getStakeholderMap(policyModel):
  smap = {}
  for stakeholder in policyModel.stakeholders:
    smap[stakeholder.name] = stakeholder
  return smap

def uniqueRecipientFull(sender, recipient):
  # if confidentiality, always unique
  if len(recipient.request.confidentiality.policies) > 0:
    return True
  else:
    return uniqueRecipient(sender, recipient)

def uniqueRecipient(sender, recipient):
  compReq = makeComparable(sender.response.integrity)
  compRec = makeComparable(recipient.request.integrity)
  combined = model.ComparableIntegrityPolicySet.join(compReq, compRec)
  checkClaims(sender.name, recipient.request.integrity.policies)
  if combined.status == model.STATUS_BOTTOM:
    return False
  if combined.status == model.STATUS_TOP:
    raise Uncombinable()
  if hasSecretPolicy(combined.policies):
    return True
  return False

def hasSecretPolicy(policies):
  for p in policies:
    if p.token == model.TOKEN_SECRET:
      return True
  return False

def uniqueSenderFull(sender, recipient):
  # if confidentiality, always unique
  if len(sender.response.confidentiality.policies) > 0:
    return True
  else:
    return uniqueSender(sender, recipient)

def uniqueSender(sender, recipient):
  compReq = makeComparable(sender.response.integrity)
  compRec = makeComparable(recipient.request.integrity)
  combined = model.ComparableIntegrityPolicySet.join(compReq, compRec)
  checkClaims(sender.name, recipient.request.integrity.policies)
  if combined.status == model.STATUS_BOTTOM:
    return False
  if combined.status == model.STATUS_TOP:
    raise Uncombinable()
  return True

def checkClaims(name, policies):
  for p in policies:
```

```
      for claim in p.claims:
        if claim.claimType == model.CLAIM_SUBJECT and claim.value != name:
          raise Uncombinable()

  def makeComparable(integritySet):
    if len(integritySet.policies) == 0:
      return model.ComparableIntegrityPolicySet(model.STATUS_BOTTOM)
    return model.ComparableIntegrityPolicySet(model.STATUS_MIDDLE, integritySet.combinator, integritySet.policies)

  class Uncombinable:
    pass
```

A.2 Demo XML files

This section includes some XML files used during development and referenced in the report.

A.2.1 service.xml

```
<?xml version="1.0" encoding="utf-8"?>
<!--
 Description: Example file for testing simple remote service
 Author: Morten Barklund
-->
<policyDocument xmlns="http://schemas.microsoft.com/wse/2003/06/Policy">
  <mappings>
    <endpoint uri="http://www.example.invalid/service.ext">
      <operation requestAction="http://www.example.invalid/some_service.ext">
        <request policy="#request" />
        <response policy="#response" />
        <fault policy="" />
      </operation>
    </endpoint>
  </mappings>>
  <policies
    xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-
                wss-wssecurity-utility-1.0.xsd"
    xmlns:wsse="http://docs.oasis-open.org/wss/2004/01/oasis-200401-
                wss-wssecurity-secext-1.0.xsd"
    xmlns:wse="http://schemas.microsoft.com/wse/2003/06/Policy"
    xmlns:wsa="http://schemas.xmlsoap.org/ws/2004/03/addressing"
    xmlns:wssp="http://schemas.xmlsoap.org/ws/2002/12/secext"
    xmlns:wsp="http://schemas.xmlsoap.org/ws/2002/12/policy"
    xmlns:wssc="http://schemas.xmlsoap.org/ws/2004/04/sc"
    xmlns:rp="http://schemas.xmlsoap.org/rp"
  >
    <wsp:Policy wsu:Id="request">
      <ALL>
        <ONEORMORE>
          <wssp:Integrity>
            <wssp:TokenInfo>
              <wssp:SecurityToken>
                <wssp:TokenType>
                  http://docs.oasis-open.org/wss/2004/01/oasis-200401-
                    wss-username-token-profile-1.0#UsernameToken
```

```
            </wssp:TokenType>
            <wssp:Claims>
              <wse:SubjectName>http://www.proxy.invalid/proxy.ext</wse:SubjectName>
            </wssp:Claims>
          </wssp:SecurityToken>
        </wssp:TokenInfo>
        <wssp:MessageParts
          xmlns:rp="http://schemas.xmlsoap.org/rp"
          Dialect="http://schemas.xmlsoap.org/2002/12/wsse#part"
        >
          wsp:Body() wse:Timestamp() wse:Header(wsp:To)
        </wssp:MessageParts>
      </wssp:Integrity>
      <wssp:Integrity>
        <wssp:TokenInfo>
          <wssp:SecurityToken>
            <wssp:TokenType>
              http://docs.oasis-open.org/wss/2004/01/oasis-200401-
                wss-x509-token-profile-1.0#X509v3
            </wssp:TokenType>
            <wssp:Claims>
              <wse:SubjectName>http://www.proxy.invalid/proxy.ext</wse:SubjectName>
            </wssp:Claims>
          </wssp:SecurityToken>
        </wssp:TokenInfo>
        <wssp:MessageParts
          xmlns:rp="http://schemas.xmlsoap.org/rp"
          Dialect="http://schemas.xmlsoap.org/2002/12/wsse#part"
        >
          wsp:Body() wse:Timestamp() wse:Header(wsp:To)
        </wssp:MessageParts>
      </wssp:Integrity>
    </ONEORMORE>
    <EXACTLYONE>
      <wssp:Confidentiality>
        <wssp:TokenInfo>
          <wssp:SecurityToken>
            <wssp:TokenType>
              http://docs.oasis-open.org/wss/2004/01/oasis-200401-
                wss-username-token-profile-1.0#UsernameToken
            </wssp:TokenType>
          </wssp:SecurityToken>
        </wssp:TokenInfo>
        <wssp:MessageParts
          xmlns:rp="http://schemas.xmlsoap.org/rp"
          Dialect="http://schemas.xmlsoap.org/2002/12/wsse#part"
        >
          wsp:Body()
        </wssp:MessageParts>
      </wssp:Confidentiality>
      <wssp:Confidentiality>
        <wssp:TokenInfo>
          <wssp:SecurityToken>
            <wssp:TokenType>
              http://docs.oasis-open.org/wss/2004/01/oasis-200401-
                wss-x509-token-profile-1.0#X509v3
            </wssp:TokenType>
          </wssp:SecurityToken>
        </wssp:TokenInfo>
        <wssp:MessageParts
          xmlns:rp="http://schemas.xmlsoap.org/rp"
          Dialect="http://schemas.xmlsoap.org/2002/12/wsse#part"
        >
          wsp:Body()
        </wssp:MessageParts>
```

```
            </wssp:Confidentiality>
          </EXACTLYONE>
        </ALL>
      </wsp:Policy>
      <wsp:Policy wsu:Id="response">
        <ALL>
          <wssp:Integrity>
            <wssp:TokenInfo>
              <wssp:SecurityToken>
                  <wssp:TokenType>
                    http://docs.oasis-open.org/wss/2004/01/oasis-200401-
                      wss-x509-token-profile-1.0#X509v3
                  </wssp:TokenType>
              </wssp:SecurityToken>
            </wssp:TokenInfo>
            <wssp:MessageParts
              xmlns:rp="http://schemas.xmlsoap.org/rp"
                Dialect="http://schemas.xmlsoap.org/2002/12/wsse#part"
            >
              wsp:Body() wse:Timestamp() wse:Header(wsp:To)
            </wssp:MessageParts>
          </wssp:Integrity>
          <EXACTLYONE>
            <wssp:Confidentiality>
              <wssp:TokenInfo>
                <wssp:SecurityToken>
                  <wssp:TokenType>
                    http://docs.oasis-open.org/wss/2004/01/oasis-200401-
                      wss-username-token-profile-1.0#UsernameToken
                  </wssp:TokenType>
                </wssp:SecurityToken>
              </wssp:TokenInfo>
              <wssp:MessageParts
                xmlns:rp="http://schemas.xmlsoap.org/rp"
                Dialect="http://schemas.xmlsoap.org/2002/12/wsse#part"
              >
                wsp:Body()
              </wssp:MessageParts>
            </wssp:Confidentiality>
            <wssp:Confidentiality>
              <wssp:TokenInfo>
                <wssp:SecurityToken>
                  <wssp:TokenType>
                    http://docs.oasis-open.org/wss/2004/01/oasis-200401-
                      wss-x509-token-profile-1.0#X509v3
                  </wssp:TokenType>
                </wssp:SecurityToken>
              </wssp:TokenInfo>
              <wssp:MessageParts
                xmlns:rp="http://schemas.xmlsoap.org/rp"
                Dialect="http://schemas.xmlsoap.org/2002/12/wsse#part"
              >
                wsp:Body()
              </wssp:MessageParts>
            </wssp:Confidentiality>
          </EXACTLYONE>
        </ALL>
      </wsp:Policy>
    </policies>
  </policyDocument>
```

A.2.2 proxy.xml

```
<?xml version="1.0" encoding="utf-8"?>
<!--
 Description: Example file for testing proxy access to remote service
 Author: Morten Barklund
-->
<policyDocument xmlns="http://schemas.microsoft.com/wse/2003/06/Policy">
  <mappings>
    <endpoint uri="http://www.example.invalid/service.ext">
      <operation requestAction="http://www.example.invalid/some_service.ext">
        <request policy="#service_request" />
        <response policy="#service_response" />
        <fault policy="" />
      </operation>
    </endpoint>
    <endpoint uri="http://www.proxy.invalid/proxy.ext">
      <defaultOperation>
        <request policy="#proxy_request" />
        <response policy="#proxy_response" />
        <fault policy="" />
      </defaultOperation>
    </endpoint>
  </mappings>
  <policies
    xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-
                wss-wssecurity-utility-1.0.xsd"
    xmlns:wsse="http://docs.oasis-open.org/wss/2004/01/oasis-200401-
                 wss-wssecurity-secext-1.0.xsd"
    xmlns:wse="http://schemas.microsoft.com/wse/2003/06/Policy"
    xmlns:wsa="http://schemas.xmlsoap.org/ws/2004/03/addressing"
    xmlns:wssp="http://schemas.xmlsoap.org/ws/2002/12/secext"
    xmlns:wsp="http://schemas.xmlsoap.org/ws/2002/12/policy"
    xmlns:wssc="http://schemas.xmlsoap.org/ws/2004/04/sc"
    xmlns:rp="http://schemas.xmlsoap.org/rp"
  >
    <wsp:Policy wsu:Id="service_request">
      <EXACTLYONE>
        <wssp:Integrity>
          <wssp:TokenInfo>
            <wssp:SecurityToken>
              <wssp:TokenType>
                http://docs.oasis-open.org/wss/2004/01/oasis-200401-
                  wss-username-token-profile-1.0#UsernameToken
              </wssp:TokenType>
              <wssp:Claims>
                <wse:SubjectName>
                  http://www.example.invalid/proxy.ext
                </wse:SubjectName>
              </wssp:Claims>
            </wssp:SecurityToken>
          </wssp:TokenInfo>
          <wssp:MessageParts
           xmlns:rp="http://schemas.xmlsoap.org/rp"
           Dialect="http://schemas.xmlsoap.org/2002/12/wsse#part">
             wsp:Body() wse:Timestamp() wse:Header(wsp:To)
          </wssp:MessageParts>
        </wssp:Integrity>
        <wssp:Integrity>
          <wssp:TokenInfo>
            <wssp:SecurityToken>
              <wssp:TokenType>
                http://docs.oasis-open.org/wss/2004/01/oasis-200401-
                  wss-x509-token-profile-1.0#X509v3
```

```
            </wssp:TokenType>
            <wssp:Claims>
              <wse:SubjectName>
                http://www.example.invalid/proxy.ext
              </wse:SubjectName>
            </wssp:Claims>
          </wssp:SecurityToken>
        </wssp:TokenInfo>
        <wssp:MessageParts
         xmlns:rp="http://schemas.xmlsoap.org/rp"
         Dialect="http://schemas.xmlsoap.org/2002/12/wsse#part">
           wsp:Body() wse:Timestamp() wse:Header(wsp:To)
        </wssp:MessageParts>
      </wssp:Integrity>
    </EXACTLYONE>
  </wsp:Policy>
  <wsp:Policy wsu:Id="service_response">
    <ALL>
      <wssp:Integrity>
        <wssp:TokenInfo>
          <wssp:SecurityToken>
            <wssp:TokenType>
              http://docs.oasis-open.org/wss/2004/01/oasis-200401-
                wss-x509-token-profile-1.0#X509v3
            </wssp:TokenType>
          </wssp:SecurityToken>
        </wssp:TokenInfo>
        <wssp:MessageParts
         xmlns:rp="http://schemas.xmlsoap.org/rp"
         Dialect="http://schemas.xmlsoap.org/2002/12/wsse#part">
           wsp:Body() wse:Timestamp() wse:Header(wsp:To)
        </wssp:MessageParts>
      </wssp:Integrity>
      <wssp:Confidentiality>
        <wssp:KeyInfo>
          <wssp:SecurityToken>
            <wssp:TokenType>
              http://docs.oasis-open.org/wss/2004/01/oasis-200401-
                wss-x509-token-profile-1.0#X509v3
            </wssp:TokenType>
          </wssp:SecurityToken>
        </wssp:KeyInfo>
        <wssp:MessageParts
          xmlns:rp="http://schemas.xmlsoap.org/rp"
          Dialect="http://schemas.xmlsoap.org/2002/12/wsse#part">
            wsp:Body()
        </wssp:MessageParts>
      </wssp:Confidentiality>
  </ALL>
  </wsp:Policy>
  <wsp:Policy wsu:Id="proxy_request">
    <wssp:Integrity>
      <wssp:TokenInfo>
        <wssp:SecurityToken>
          <wssp:TokenType>
            http://docs.oasis-open.org/wss/2004/01/oasis-200401-
              wss-username-token-profile-1.0#UsernameToken
          </wssp:TokenType>
        </wssp:SecurityToken>
      </wssp:TokenInfo>
        <wssp:MessageParts
         xmlns:rp="http://schemas.xmlsoap.org/rp"
         Dialect="http://schemas.xmlsoap.org/2002/12/wsse#part">
           wsp:Body() wse:Timestamp() wse:Header(wsp:To)
        </wssp:MessageParts>
```

```
        </wssp:Integrity>
      </wsp:Policy>
      <wsp:Policy wsu:Id="proxy_response">
        <wssp:Integrity>
          <wssp:TokenInfo>
            <wssp:SecurityToken>
              <wssp:TokenType>
                http://docs.oasis-open.org/wss/2004/01/oasis-200401-
                  wss-username-token-profile-1.0#UsernameToken
              </wssp:TokenType>
            </wssp:SecurityToken>
          </wssp:TokenInfo>
            <wssp:MessageParts
             xmlns:rp="http://schemas.xmlsoap.org/rp"
             Dialect="http://schemas.xmlsoap.org/2002/12/wsse#part">
                    wsp:Body() wse:Timestamp() wse:Header(wsp:To)
            </wssp:MessageParts>
        </wssp:Integrity>
      </wsp:Policy>
    </policies>
  </policyDocument>
```

Bibliography

[1] Arindam Banerji, Claudio Bartolini, Dorothea Beringer, Venkatesh Chopella, Kannan Govindarajan, Alan Karp, Harumi Kuno, Mike Lemon, Gregory Pogossiants, Shamik Sharma, and Scott Williams. Web Services Conversation Language (WSCL) 1.0, 2002. http://www.w3.org/TR/wsdl20-adjuncts/, [Online; accessed June-1-2007].

[2] Bruno Blanchet. An Efficient Cryptographic Protocol Verifier Based on Prolog Rules. In *14th IEEE Computer Security Foundations Workshop (CSFW-14)*, pages 82–96, Cape Breton, Nova Scotia, Canada, June 2001. IEEE Computer Society.

[3] C. Bodei, M. Buchholtz, P. Degano, H. Riis Nielson, and F. Nielson. Automatic validation of protocol narrations. In *16th IEEE Computer Foundations Workshop (CSFW 03). pp 126-140*, 2003. IEEE Computer Society Press.

[4] David Booth and Canyang Kevin Liu. Web Services Description Language (WSDL) Version 2.0 Part 0: Primer, 2007. http://www.w3.org/TR/wsdl20-primer/, [Online; accessed June-1-2007].

[5] Don Box, David Ehnebuske, Gopal Kakivaya, Andrew Layman, Noah Mendelsohn, Henrik Frystyk Nielsen, Satish Thatte, and Dave Winer. Simple Object Access Protocol (SOAP) 1.1, 2000. http://www.w3.org/TR/2000/NOTE-SOAP-20000508/, [Online; accessed June-1-2007].

[6] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler, François Yergeau, and John Cowan. Extensible Markup Language (XML) 1.1 (Second Edition), 2006. http://www.w3.org/TR/xml11/, [Online; accessed June-1-2007].

[7] Mikael Buchholtz. User?s Guide for the LySatool version 2.01, April 2005.

[8] Roberto Chinnici, Hugo Haas, Amelia A. Lewis, Jean-Jacques Moreau, David Orchard, and Sanjiva Weerawarana. Web Services Description Language (WSDL) Version 2.0 Part 2: Adjuncts, 2007. http://www.w3.org/TR/wsdl20-adjuncts/, [Online; accessed June-1-2007].

[9] Roberto Chinnici, Jean-Jacques Moreau, Arthur Ryman, and Sanjiva Weerawarana. Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language, 2007. http://www.w3.org/TR/wsdl20/, [Online; accessed June-1-2007].

[10] Erik Christensen, Francisco Curbera, Greg Meredith, and Sanjiva Weerawarana. Web Services Description Language (WSDL) 1.1, 2001. http://www.w3.org/TR/wsdl, [Online; accessed June-1-2007].

[11] John Cowan and Richard Tobin. XML Information Set (Second Edition), 2003. http://www.w3.org/TR/xml-infoset/, [Online; accessed June-1-2007].

[12] D. Dolev and A.C. Yao. On the security of public key protocols. In *22nd Annual Symposiumon Foundations of Computer Science, pp 126-140*, 1981. IEEE.

[13] Martin Gudgin, Marc Hadley, Noah Mendelsohn, Jean-Jacques Moreau, and Henrik Frystyk Nielsen. SOAP Version 1.2 Part 1: Messaging Framework, W3C Recommendation, 2004. http://www.w3.org/TR/soap12-part1/, [Online; accessed June-1-2007].

[14] Martin Gudgin, Marc Hadley, Noah Mendelsohn, Jean-Jacques Moreau, and Henrik Frystyk Nielsen. SOAP Version 1.2 Part 2: Adjuncts, W3C Recommendation, 2004. \url{http://www.w3.org/TR/soap12-part2/}, [Online; accessed June-1-2007].

[15] IBM and Microsoft. Web Services Inspection Language, 2007. http://www-128.ibm.com/developerworks/library/specification/ws-wsilspec/, [Online; accessed June-1-2007].

[16] IMM. LySaTool, 2003. http://www2.imm.dtu.dk/cs_LySa/lysatool/; accessed June-1-2007.

[17] Microsoft. Web Services Enhancements (WSE), 2007. https://msdn2.microsoft.com/en-us/library/aa139633.aspx, [Online; accessed June-1-2007].

[18] Cambridge Microsoft Research. Samoa: Formal tools for securing web services, 2007. http://research.microsoft.com/projects/samoa/, [Online; accessed June-1-2007].

[19] Nilo Mitra and Yves Lafon. SOAP Version 1.2 Part 1: Primer (Second Edition), W3C Recommendation, 2004. http://www.w3.org/TR/soap12-part0/, [Online; accessed June-1-2007].

[20] Microsoft Research. Tulafale: A security tool for web services, 2006. http://research.microsoft.com/research/downloads/Details/a91c6322-ae04-4b7c-9f8b-908f094d7a15/Details.aspx, [Online; accessed June-1-2007].

[21] Wikipedia. SOAP — Wikipedia, The Free Encyclopedia, 2007. http://en.wikipedia.org/w/index.php?title=SOAP&oldid=134358638, [Online; accessed June-1-2007].

[22] Wikipedia. Web Services Description Language — Wikipedia, The Free Encyclopedia, 2007. http://en.wikipedia.org/w/index.php?title=Web_Services_Description_Language&oldid=125733132, [Online; accessed June-1-2007].