IT University of Copenhagen

Bachelor Project

Verifiable Secure Open Source Alternative to NemID

Authors:

Andreas Hallberg Kjeldsen
ahal@itu.dk

Morten Chabert Eskesen
mche@itu.dk

Supervisor:

Dr. Joseph Roland KINIRY
josr@itu.dk

Co-supervisor:

Hannes Mehnert
hame@itu.dk

May 22, 2013

Abstract

Denmark currently has NemID to provide every resident of Denmark with a digital signature. NemID is an authentication application written in Java. The Java applet is used by banks in Denmark and other government institutions for identifying users. Lately there has been many security issues involved with the Java browser plug-in. This report's objective is to create a replacement for NemID that is both verifiable secure and open source. We propose the use of executable models to prove the security properties of such a system. We provided a detailed the description of the protocol which is based on the SAML Single Sign-On protocols. We use the programming language F* which is a value-dependent, typed language with refinement types to formalize the protocol. We have formalized key parts of the protocol therefore proved it possible to formalize such protocols as a whole with F*. Therefore we have shown that it is possible to use a language like F* to formally specify the protocol and have an executable model at the same time.

Acknowledgements

First of all we would like to thank Joseph Roland Kiniry for his interest in our work and for his many encouraging and helpful comments. Secondly we would also like to thank Hannes Mehnert for his help in designing the protocol and his insights. Lastly we would like to thank Jacob Højgaard for laying the ground work in his masters thesis *Securing Single Sign-On System With Executable Models.*

Contents

List of Figures

Listings

Chapter 1

Introduction

Security through obscurity is a principal within security engineering. It refers to the reliance on secrecy of the design or implementation of a system to provide security. Kerckhoffs's principle dictates that a cryptographic system where only the key is kept private should be secure even in the hands of the enemy.

> *"There is no security by obscurity"* - Auguste Kerckhoffs [6]

Based on this, it can be assumed that in the making of a secure system obscurity should not be used for improving the security.

> *"A system is secure only when it is secure in the light of day, under full public view."* - Dr. Joseph Roland Kiniry [21]

In Denmark it is required to use NemID when accessing public websites, online banks and other services requiring a digital signature (DanskeSpil e.g.). NemID claims to be a secure system, they state various reason that would give the impression that they are right [22]. But looking at the system from the outside, there is no way of telling if they're right.

Even though there is no public record of the NemID system being completely compromised, there is no way for us outsiders to verify whether or not it could happen. This is due to NemID following the security through obscurity principal. The source code for their key components are not publicly available, the Java applet they are providing has been obfuscated and the API they have made available does not reveal enough to conclude anything.

1.1 Objectives

In this project we set out to create a replacement for NemID that is both verifiable secure and also open source. For the sake of being able to reference this new system throughout the report, we will refer to it as *OpenNemID*.

The goal of this project is to:

1. Describe and outline the OpenNemID protocol, including but not limited to registration and login.
2. Formalize the specification of OpenNemID in F* to the extent possible.

1.2 Scope

This project has had its focus towards specifying a new protocol that could replace NemID. The intent of this project is therefore not to develop a complete system, but to make a formal specification for a system that could then later be used to engineer a new system.

1.3 Background

We're extending the work done by Jacob Højgaard in his Masters thesis *'Securing Single Sign-On Systems With Executable Models'*. Jacob's research has focused on the current implementation of NemID and therefore describes, outlines and models the current system used in Denmark as of May 2013.

NemID is a part of NemLog-in, which is owned by the Danish government through the Agency of Digitalization, henceforth *DIGST*. NemLog-in is a key component in the ambitious digitalization strategy outlined by the DIGST [18]. The digitalization strategy specifies that all interaction between residents and any public institution should be possible online from 2015. A public login federation[1], NemLog-in, is used for this purpose.

Nets DanID[2], has developed NemID. Their contract runs from august 21st 2008 to august 20th 2015, though the contract can be extended twice, each time by a one year period [19]. When the current contract expires it is possible that a new contract mandating a replacement for NemID could be established.

NemID have received quite a lot of criticism because it uses Java as the frontend for its users. Java is cross-platform (Windows, OS X, Unix etc.) and can be embedded directl onto webpages. A reason for choosing Java for NemID, could be its cross-platform capabilities, but it is more likely that is was

[1]Login federation refers to the concept of federated identity where a user can link identities stored in different identity management systems. The term *Federation* refers to a union of selfgoverning states, which is very applicable for this term [1].

[2]Nets DanID A/S is a company within the Nets group, which is owned by the danish national bank and a number of danish banks

the cryptographic features (applet signing e.g.) that made the difference. Unfortunately Java is one of the most frequently exploited pieces of software [23], which in turn means that to use NemID, the user would have to install software that later could result in the user's computer being compromised. Furthermore, most mobile platforms has no support for Java, thereby rendering it impossible to use NemID on a mobile device.

From an academic point of view it would be interesting to see if a platform independent replacement for NemID can be made by students within a very limited timeframe, that does not rely on hidden and obfuscated code, but on open source and verification tools to prove the system secure.

Chapter 2

Technical background

This chapter will clarify the technical concepts that has been used in this report. It will describe the SAML protocol and the Danish specialization OIOSAML. Furthermore it will describe the concept of static analysis and outline different protocol verification tools and our choice of tool. Lastly it will introduce the concept *N-factor authentication* that has been used in the development of this project.

2.1 SAML protocol

The Security Assertion Markup Language or SAML is an XML based language created for the exchanging of authentication and authorization (security tokens) between different systems or domains, in particular between a service provider and an identity provider. The latest version of SAML is 2.0 released in 2005. SAML specifies three roles: *principal*, *identity provider* and *service provider*. Principal refers to an entity that can be authenticated, therefore both the user and the identity provider and service provider are principals. SAML addresses the problem of web browser single-sign on (SSO). SSO means that the user only has to login once per identity provider per session. Practically this means that the user only has to be redirected once to the identity provider for authentication per session. The identity provider creates a session for the user that can be reused for any subsequent authentication request. SAML specifies 5 core elements - *Assertions*, *Protocols*, *Bindings*, *Profiles* and *Metadata*. See appendix A for examples of assertion and SAML messages.

Assertions are security tokens containing statements or claims about a principal. These statements in SAML are called attributes and they usually hold information like name, last name, email etc. The principal is referred to as the *subject* in an assertion.

$$\begin{aligned}
Assertion := [&Id : String\\
&IssueInstant : DateTime\\
&Issuer : String\\
&Signature : String\\
&Subject : Subject\\
&Conditions : Conditions\\
&AttributeStatement : List < attribute >\\
&AuthnStatement : AuthnStatement]\\
\\
Subject := [&NameId : String\\
&InResponseTo : String\\
&NotOnOrAfter : DateTime\\
&Recipient : String]\\
\\
Conditions := [&NotBefore : DateTime\\
&NotOnOrAfter : DateTime\\
&AudienceRestriction : List < String >]\\
\\
AuthnStatement := [&AuthnInstant : DateTime\\
&SessionIndex : String]
\end{aligned}$$

Figure 2.1: The elements in a SAML Assertion

Protocols describe the messages that can be exchanged between the service provider and the identity provider when exchanging *assertions*. The protocol used in OpenNemID is called the Authentication Request Protocol. This protocol consists of AuthnRequest message and a response message. The AuthnRequest message has an id that must be unique and the creator of the message is responsible for ensuring it is. AuthnRequest has 5 fields:

1. The *Id* - unique identifier for the message.
2. *IssueInstant* - time stamp from when the message was issued.
3. The *Destination* - the receiver of the request.
4. *Issuer* - the initiator of the authentication.
5. *Conditions* - conditions of the request (audience restriction, not before this time or not after this time).

$$\begin{aligned}
AuthnRequest := [&Id : String\\
&IssueInstant : DateTime\\
&Destination : URI\\
&Issuer : String\\
&Conditions : Conditions]
\end{aligned}$$

Figure 2.2: The SAML AuthnRequest message

The response message has fields *Id* and *IssueInstant* that was specified in the request message. The *Destination* is the endpoint address for the service provider. The *Issuer* is the entity Id of the identity provider. The response message also has a field *inResponseTo* that specifies the id of the AuthnRequest the response is to. The *Status* field can only have one of three possible values - *Requester*, *Responder* or *Success*. The *Assertion* can only be issued if the *Status* field is *Success*. The assertion can either be plain text or an *EncryptedAssertion* which will be encrypted under the public key of the recipient of the message (service provider).

$$
\begin{aligned}
Response := [&Id : String\\
&IssueInstant : DateTime\\
&Destination : URI\\
&InResponseTo : String\\
&Issuer : String\\
&Status : String\\
&Assertion : Assertion]
\end{aligned}
$$

Figure 2.3: The SAML Response message

Bindings specifies how the messages are mapped to the underlying HTTP(S) or SOAP protocols. However in this report only HTTP POST and HTTP REDIRECT bindings will be addressed. The POST binding maps the content of messages to hidden XHTML form fields that are named SAMLRequest and SAMLResponse. The content is BASE64 encoded. For the REDIRECT binding the message content is mapped to the URL query string with SAMLRequest and SAMLResponse as the identifier. Since the URL query string has limited capacity the message is compressed by the DEFLATE algorithm and after that it is BASE64 encoded and URL encoded.

Profiles specifies how the *assertion*, *protocol* and *binding* are used to fulfill a specific requirement or a use case. The Danish specialization of the SAML Web Browser SSO Profile, OIOSAML, will be described in further detail in section 2.1.1.

Metadata is the necessary data exchanged between the involved parties in order to know each other. A service provider's metadata contains the certificate for signing messages and a message that the identity provider should use for encrypting assertion. It also contains endpoints which specifies the addresses to which the identity provider should send response messages. There can be more than one endpoint if different bindings are available. The identity provider's metadata also contains signing and encryption certificates and endpoints that specify the addresses for sending requests to the identity provider. There can be more endpoints for different bindings. The identity provider will publish the attributes it is able to serve through it's metadata.

2.1.1 OIOSAML

The Danish specialization OIOSAML, *OIO Web SSO Profile*, is a dialect of the SAML profile *Web Browser SSO*. This profile is a standard that specifies the interactions between a user and the government and/or other public institutions.

OIOSAML has specific requirements for binding and protocol. The point of adding these measures is to secure the protocol further and that some of the extensibility of the SAML Web Browser SSO is not necessary. An example is that in plain SAML it is possible to issue authentication requests without having a prior exchange of metadata. This is not desirable in OpenNemID since the security of the protocol will be improved by adding a controlled connection flow, where the service providers are approved beforehand. In the following there has been some concepts left out if deemed insignificant to the level of description necessary for this report.

OIOSAML mandates for the Service Provider

- The metadata of the service provider should be known to the identity provider (as mentioned before there must be a formal exchange of metadata between the parties)
- Binding for *AuthnRequest* should be HTTP REDIRECT with the message DEFLATE encoded
- Transport should be over (one-way) SSL/TLS
- The *AuthnRequest* must be signed and the signature placed as a query parameter identified as "Signature"
- The RelayState[1] must be opaque

OIOSAML mandates for the Identity Provider

- The metadata of the service provider should be known to the identity provider (as mentioned before there must be a formal exchange of metadata between the parties)
- If the response is an error response the identity provider must not include any assertions
- Successful responses can contain only one assertion
- The assertions must state the level of authentication achieved[2]
- Response messages should be sent to the service provider using the HTTP POST binding

[1] RelayState is a mechanism that the service providers can use to associate any subsequent profile messages with the original request (could be the protected resource requested by the user)

[2] Authentication level refers to the level of authentication achieved by the subject on a scale from 1-5 where 5 is highest.

- Transport should be over (one-way) SSL/TLS
- Response messages should not be signed but instead the embedded *Assertion* should be signed and encrypted

OIOSAML mandates for assertions

- Assertions must contain only one AuthnStatement and one AttributeStatement
- The assertion must be encrypted with the certificate received from the service provider through the exchanged metadata
- The issuer field which is the entity Id of the identity provider must be included in the assertion
- The assertion must be signed with the certificate described in the metadata exchanged with the service provider

2.2 Static analysis

Static program analysis is the opposite of dynamic analysis which is analysis performed on executing programs. Therefore static analysis is analysis of computer software that is performed without actually executing programs. Static analysis is usually the analysis performed by an automated tool. There exists tools to perform analysis on the behavior of individual statements and declarations and tools to perform analysis of the complete source code of a program. Static analysis has been successfully used to automatically validate security properties of classical protocols. The technique has been used to validate the SAML Single Sign-On protocols [3].

2.3 Protocol verification tools

A communication protocol is a system of digital message formats and rules for exchanging those messages in or between computing systems. A cryptographic protocol or security protocol is an abstract or concrete communications protocol that performs a security-related function and applies cryptographic models. Cryptographic protocols can be verified formally on an abstract level. Formal verification is an attempt at proving or disproving the correctness of intended algorithms underlying a system with regards to a certain formal specification. There exists various tools to formally verify a cryptographic protocol on an abstract level. This section will outline the tools we have investigated through the work of this project and the key features of these tools.

2.3.1 ProVerif

ProVerif is an automatic cryptographic protocol verifier in the Dolev & Yao model[3]. It is based on a representation of the protocol by Horn clauses[4]. One of ProVerif's key features is its ability to handle many different cryptographic primitives, including shared- and public-key cryptographys, hash functions and Diffie-Hellman key agreements[5]. The other key feature is its ability to handle an unbounded number of sessions of the protocol (even in parallel). ProVerif can prove secrecy[6], authentication, strong secrecy[7] and equivalences between processes that differ only by terms.

2.3.2 F*

F* is a new research language from Microsoft Research. The language is dependently typed for secure distributed programming. F*'s purpose is the enabling of the construction and communication of proofs of program properties and secondly of properties of a program's environment in a verifiable secure way. F* is based on the functional programming language F# and compiles to .NET bytecode in a type preserving style. Even though F* is formal specification language it is still executable. Furthermore F* has a fully abstract compiler from F* to JavaScript. The current version of the F* compiler is considered to be an α-release.

2.3.3 JSCert

JSCert is an ongoing project. JSCert is short for Certified JavaScript. The goal of the project is to really understand JavaScript. The project is building models of ECMAScript[8] semantics in the Coq proof assistant and automated logical reasoning tools built on those semantics. The project introduces a program logic for reasoning about a broad subset of JavaScript. They have proved a strong soundness result. The libraries written in their subset and proved correct with respect to their specifications will be well-behaved - even when called by arbitrary JavaScript code.

2.3.4 XOR-ProVerif

XOR-ProVerif is a tool for analyzing protocols with XOR. It is a tool with an implementation of the reduction of the derivation problem for Horn theories with XOR to the XOR-free case in combination with ProVerif. These theories

[3]Cryptographic primitives are assumed perfect and cyphers cannot be decrypted without the proper decryption key

[4]Horn clause is a clause (a disjunction of literals) with at most one positive literal

[5]Diffie-Hellman keyexchange is a specific method for the exchanging of cryptographic keys

[6]The adversary cannot obtain the secret

[7]The adversary does not see the difference when the value of the secret changes

[8]ECMAScript is a scripting language often used for client-side scripting on the web

allow the modeling of a large class of intruders capabilities and protocols that employ the XOR operator.

2.3.5 DH-ProVerif

DH-ProVerif is a tool for analyzing protocols with the Diffie-Hellman exponentiation. The tool is an adaption of the XOR-ProVerif approach to the case of Diffie-Hellman exponentiation. The approach is to reduce the derivation problem for Horn theories modulo algebraic properties of Diffie-Hellman exponentiation to a purely syntactical derivation problem for Horn theories. The reduction for Diffie-Hellman exponentiation is more efficient than the one for XOR.

2.3.6 CSProto

CSProto is a tool for analyzing contract signing protocols. The tool is used to prove that certain game-theoretic security properties, including balance for contract signing protocols, can be decided in a Dolev & Yao style model with a bounded number of sessions. The decision algorithm is based on constraint-solving procedures which have been successfully employed in tools for reachability properties.

2.3.7 Our choice

We have chosen to formally specify OpenNemID using F* for several reasons. We both have previous experience working with the functional programming language F# which F* is based on and a little experience with ML. However the most contributing factors was:

1. Since we are extending the work done by Jacob it made sense to use F* as he did
2. The fully abstract compilation from F* to JavaScript
3. F* is executable

2.4 N-factor authentication

Two-factor authentication is the process of a requesting entity presenting some evidence of its identity to a second identity. The goal of two-factor authentication is to decrease the possibility that the requester is presenting false evidence of its identity. In a login scenario the requester would be a user requesting to login. Two-factor authentication requires the use of two of three authentication factors:

1. Something the requester knows (e.g. password)

2. Something the requester has (e.g. mobile phone)
3. Something the requester is (e.g. biometric characteristic - such as fingerprint)

NemID currently implements two-factor authentication. It uses the factor 1 which is the user's userid and password. It also uses factor 2 which is the physical cardboard keycard that comes with creating a NemID login. This cardboard keycard comes with numeric challenges and their corresponding numeric keys. So when a user logs in with their userid and password they will be prompted for the numeric key corresponding to the numeric challenge displayed in the Java applet.

The OpenNemID protocol extends the two-factor authentication to *n-factor authentication*, thereby adding multiple layers of security that would make it harder for an adversary to login. The first authentication is the factor 1 - userid and password. After this there are various ways for a user to authenticate themselves further based on their registration. However it will only use either the factor 1 or factor 2. Even though it would be preferable with a biometric characteristic authentication factor for digital identities, this is at the moment impossible. The OpenNemID protocol requires that a user at least has the standard factor 1 (a userid and a password) and at least one more authentication method. The user will by default have to authenticate themselves through all the authentication methods they have added.

The authentication methods that has been considered are:

OpenID
OpenID is an open standard federated login utilizing co-operating sites (or relying parties) for user authentication. OpenID is not limited to a single institution provided the identity of the users, instead there can be multiple providers, some of the major ones are Google, Yahoo, VeriSign and Wordpress [25]. To use OpenID the users would be required to create and setup an account with one of the providers.
Some of the providers might have introduced extra security step, one such provider is Google. Google has its own security implementations, such as two-factor authentication. As OpenNemID would depend on the user fully authenticating with OpenID, the user would also have to pass Googles two-factor authentication in the case of authorizing at Google.

Facebook Connect
Facebook Connect is the single sign-on solution provided by Facebook. This would require the user to create and setup a Facebook account.

One-Time Passwords
A One-Time Password, henceforth *OTP*, is a generated password with the limitation of only being usable once. To further enhance the security of the OTP, a time limiting factor could be applied, thereby creating a Time-based One-time Password, henceforth *TOTP*.

A TOTP is generated based on a uniform algorithm, the requirements for the algorithm includes that the key must be shared by the prover and the verifier. The time-step value[9] must be the same for the prover and the verifier, both the prover and the verifier must have the capabilities of deriving the current Unix timestamp[10] [24].
Various client applications have been developed for generating the TOTP. *Google Authenticator* is Google's open-source TOTP client that runs on mobile platforms such as Android, iOS and Blackberry.

Yubico Yubico has specialized in OTP hardware and software. Their flagship product, the YubiKey, is a tiny device that is inserted into the USB port on the user's device. When challenged with a OTP, the user would use the YubiKey to generate the OTP. YubiKey is already integrated with various single-sign on systems such as OpenID.

SMS
An OTP could be generated and send to the user's mobile phone.

Knowledge-based Authentication
Knowledge-based authentication can either by static or dynamic.
Static knowledge-based authentication refers to predefined questions and answers, this could be a question such as "What is your mother's maiden name" and the user would have input the proper answer beforehand. For our implementation it is not specified whether or not the questions should be predefined by the authentication provider or if the user should be able to create them.
Dynamic knowledge-based authentication refers to questions being generated on the fly based on data stored related to the user, this could be questions like "What street did you live on at your previous residence?".

[9]The time-step value is the amount of seconds that must be between each generated password. If the first password was generated at 01:03:02 PM and the time-step value was 30, the next password must be generated at 01:03:32 PM.

[10]Unix time, or POSIX time, refers to the amount of seconds that has passed since January 1st 1970 00:00:00 UTC, without counting leap seconds.

Chapter 3

Modelling the protocol

Before formalizing the protocol, it's required to specify and explain some of the words, concepts and meanings used within the protocol. This will be done by using graphical representations of the message flow.

3.1 A protocol walkthrough

A sequence diagram depicting the protocol is useful for introducing the basics of the protocol. The sequence diagrams give a brief overview of the logon protocol, identity provider registration protocol and the user registration protocol. The logon protocol will, later in this chapter, be modeled using a communication diagram, this is done to give a more thorough descriptions

3.1.1 Logon protocol

Jacob has depicted the NemID protocol with 12 steps. The sequence diagram shows how the browser must delegate messages from the service provider to the identity provider, while also having to delegate messages from DanId to the identity provider. A single challenge is used to authenticate the user after credentials have been submitted.

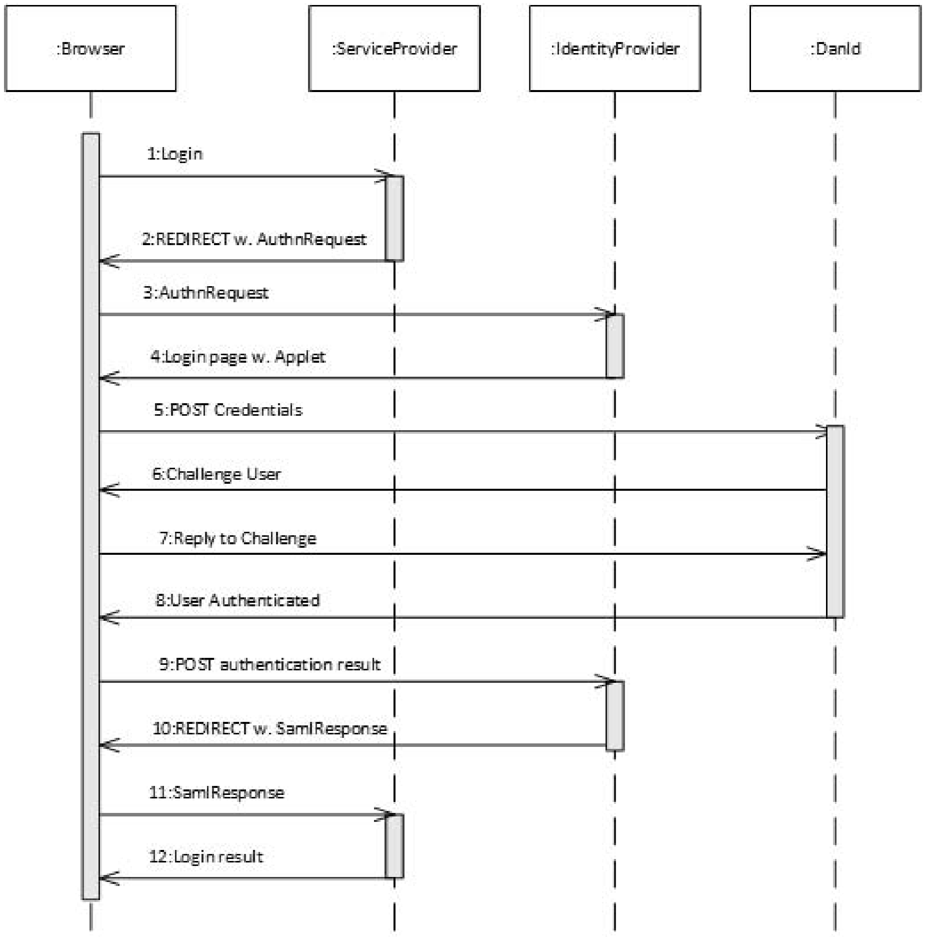

Figure 3.1: Sequence diagram of authentication with NemID [1]

In comparison to NemID, the OpenNemID protocol have modified the way messages are sent. The amount of messages the browser must delegate has been limited to a few. There is not just one challenge used to authenticate the user, there are n challenges, as described in section 2.4. The amount of steps in the protocol have been reduced to 8.

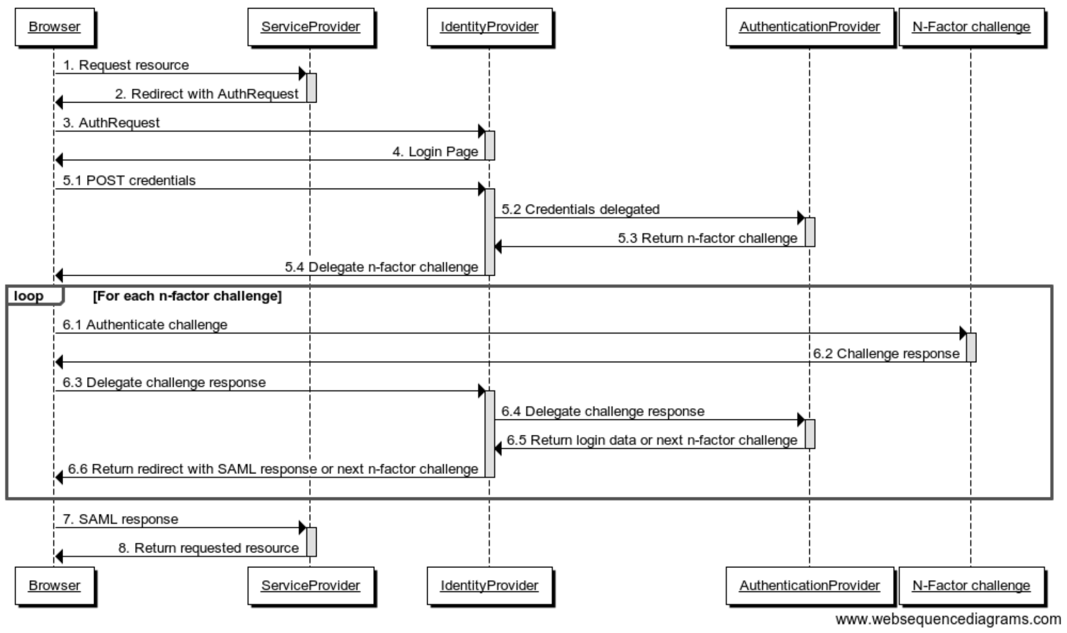
Browser
ServiceProvider
IdentityProvider
AuthenticationProvider
N-Factor challenge
1. Request resource
2. Redirect with AuthRequest
3. AuthRequest
4. Login Page
5.1 POST credentials
5.2 Credentials delegated
5.3 Return n-factor challenge
5.4 Delegate n-factor challenge
loop
[For each n-factor challenge]
6.1 Authenticate challenge
6.2 Challenge response
6.3 Delegate challenge response
6.4 Delegate challenge response
6.5 Return login data or next n-factor challenge
6.6 Return redirect with SAML response or next n-factor challenge
7. SAML response
8. Return requested resource
www.websequencediagrams.com

Figure 3.2: Sequence diagram of authentication with OpenNemID

3.1.2 Identity provider registration protocol

An identity provider has to register with an authentication provider to establish a trusted relationship and an agreement that the authentication provider will in fact act as a provider for authenticating users. A challenge will be send to the identity provider to confirm that the identity provider is who they claim to be.

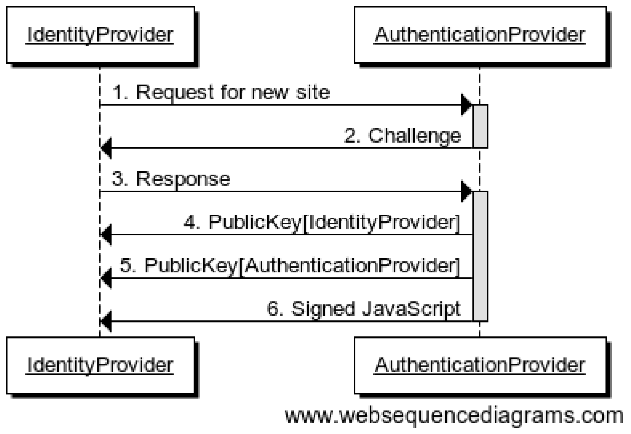

Figure 3.3: Sequence diagram of identity provider registration with OpenNemID

3.1.3 User registration protocol

Before a user is able to authenticate at the authentication provider, they have to register there first. The user has to activate their account using credentials received via mail. Upon activation, a new password must be chosen, optionally a userid can also be chosen. The user must be able to add and revoke n-factor challenges while also being able to revoke previously granted identity provider permissions.

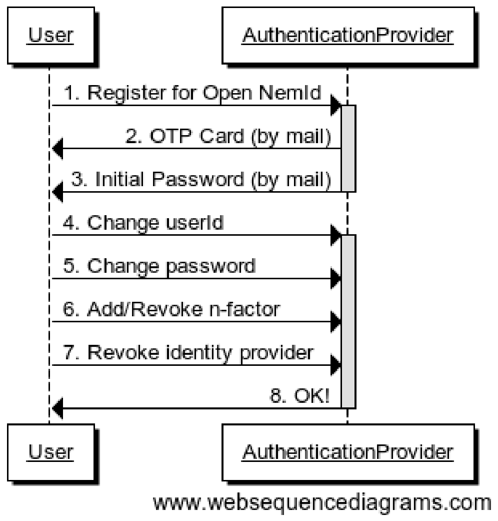

Figure 3.4: Sequence diagram of user registration with OpenNemID

3.2 Protocol prerequisites

It's important to have some requirements as to how the systems should function. The requirements helps define certain properties the involved participants must have or obey to.

3.2.1 Shared

The NemLog-in specification mandates the use of OIOSAML, this will most likely not be excluded, therefore we assume that OpenNemID also has to use it. Further OIOSAML mandates the use of one-way SSL/TLS for all bindings, the mandate does not specify a specific version, though it can be assumed that a minimum version of SSL 3.0 due to the fact that SSL 2.0 is in general considered deprecated. We assume the use of SSL 3.0 or TLS 1.0 in this report.

As mentioned before, SAML uses the browser to transfer messages from one principal to the other. The way to do this is through HTTP REDIRECTs, which could be either a HTTP-GET REDIRECT or a HTTP-POST REDIRECT. The HTTP protocol accepts a Location head in the HTTP RESPONSE which indicates where the browser should redirect to. The location header redirect will act as a HTTP-GET REQUEST which excludes the usage of POST data, thereby limiting the amount of data that can be transferred. To overcome this problem HTTP-POST REDIRECTs are used, these are not a part of the HTTP protocol, but is synthesized by using JavaScript to emulate a regular HTTP-POST REQUEST. Therefore it is required for the users browser to follow redirects and to have JavaScript enabled.

For there to be any actual messages to flow between the service provider and identity provider, it is assumed that they reside in different domains and are different entities.

The identity provider is only to issue assertions to known service providers, this requires that SAML metadata has been changed beforehand. The certificates used for signing and encrypting has to be checked for revocation and validity whenever used.

To summarize:

1. OIOSAML mandates the use of SSL(3.0)/TLS(1.0).
2. The browser must follow redirects.
3. The browser must have JavaScript enabled.
4. Service provider and identity provider are different entities residing in different domains.
5. SAML metadata must have been exchanged between the service provider and identity provider.
6. Signature check and encryption requires validity/revocation check of the certificate.

3.2.2 NemID specifics

It's required for the OCES certificates used for signing and encrypting to have been issued by DanID.

3.2.3 OpenNemID specifics

For the communication between the authentication provider and the identity provider, a secure tunnel must have been set up. Further the user must have registered at the authentication provider.

We have assume the availability of a web cryptography API in this report. A web cryptography API has not yet been standardized, but a standardization is being worked on, a draft is available [10].

3.3 Formalizing protocol messages

The UML communication diagrams depicting the protocols are made up of two or more participants, henceforth principals, and the messages flowing through the system. The line between two principals indicates a channel where communication can flow, this channel is assumed to be a secure channel, meaning for HTTP messages, the HTTPS protocol would be used. An arrow indicates the direction of the message and the text on top of the arrow is the message being sent. The messages does not conform to any specific formalism, but follows a simple syntax. Messages are, very applicable, named in accordance with their HTTP protocol verb. The messages are to be interpreted the following way:

GET means a HTTP-GET request, the parameter is the resource being requested.

POST means a HTTP-POST request, the parameters are the destination for the request followed by the data being submitted.

REDIRECT means either a HTTP-REDIRECT or a JavaScript redirect, whichever is used is not important for the purpose of the description. The parameters are the destination for the redirect followed by the parameters to include in the redirect.

RESPONSE means a HTTP-RESPONSE messages. The parameters are either the data included in the response or a HTTP status code indicating the type of the response along with a message, this is used for indicating when an error happen.

DELEGATE means forwarding the data from the previous request, the parameters are the parameters from the previous request that were to be delegated.

AUTHENTICATE is to be interpreted as the sequence of actions required to be authenticated at the specified NFactorChallenge. The AUTHENTICATE message is defined this generically on purpose, as the way a user would authenticate for a NFactorChallenge can vary a lot depending on the challenge (see Section 2.4).

3.4 Communication model

To make the changes from NemID to OpenNemID clear, we will first show the communication diagram for NemID, afterwards we will show the communication diagram for OpenNemID. Both diagrams make use of abbreviations, these abbreviations are listed at the top of the diagram along the word or phrase they abbreviate. We have tried to comply with Jacobs data as much as possible, due to the fact that his report conforms to the requirements mandated by DIGST.

3.4.1 Message processing

Information regarding the processing of messages have not been included in the diagrams, this is to prevent cluttering of the diagrams. To circumvent this, the processing rules will be described afterwards. The descriptions will be linked to a specific process number, meaning Process 3 would be the handling and response of message 3 in the diagram. Messages that are self-explanatory will not be further described.

3.4.2 Communication diagram for NemID

Description of the message processing for this diagram has been left out of our report, they can however be found in Jacob Højgaards report [1].

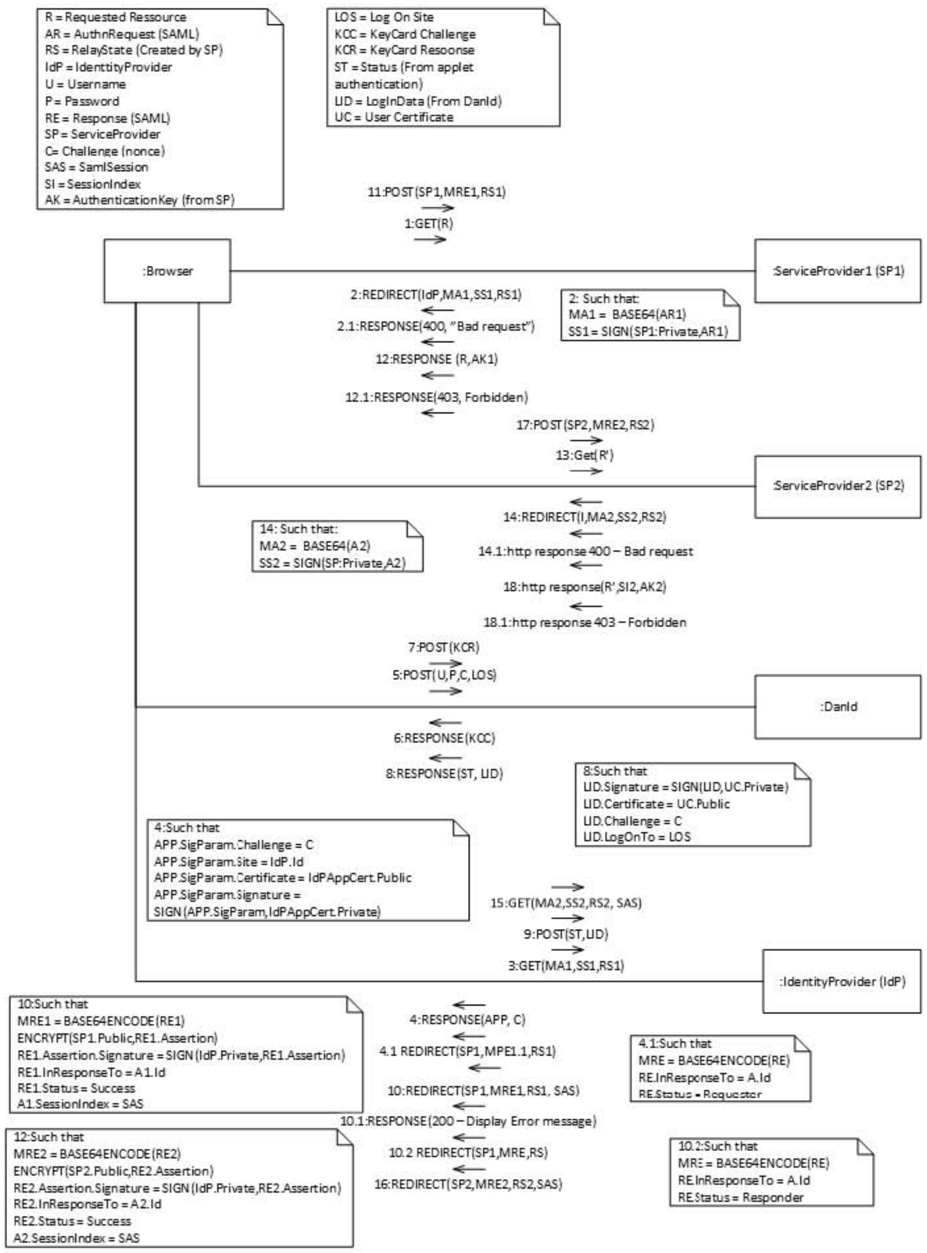

Figure 3.5: Communication diagram for the complete NemID protocol [1]

3.4.3 Communication diagram for OpenNemID

In this diagram, we have chosen to leave out the additional request to another service provider than the one initially used, this is due to the communication flow being exactly the same as in Jacobs diagram, see Figure 3.5, message 13 to 18.

In the communication diagram for OpenNemID, DanID have been replaced by AuthenticationProvider, for the sake of our protocol it is of no greater importance which company handles the authentication.

We have strived to minimize the amount of messages flowing through the system to limit the amount of possible attack points for a potential adversary. We have also eliminated the need for transporting sensitive data, such as the users login data, from the authentication provider to the identity provider, by mandating that the identity provider and the authentication provider exchange information using a secure tunnel without the user being able to interfere.

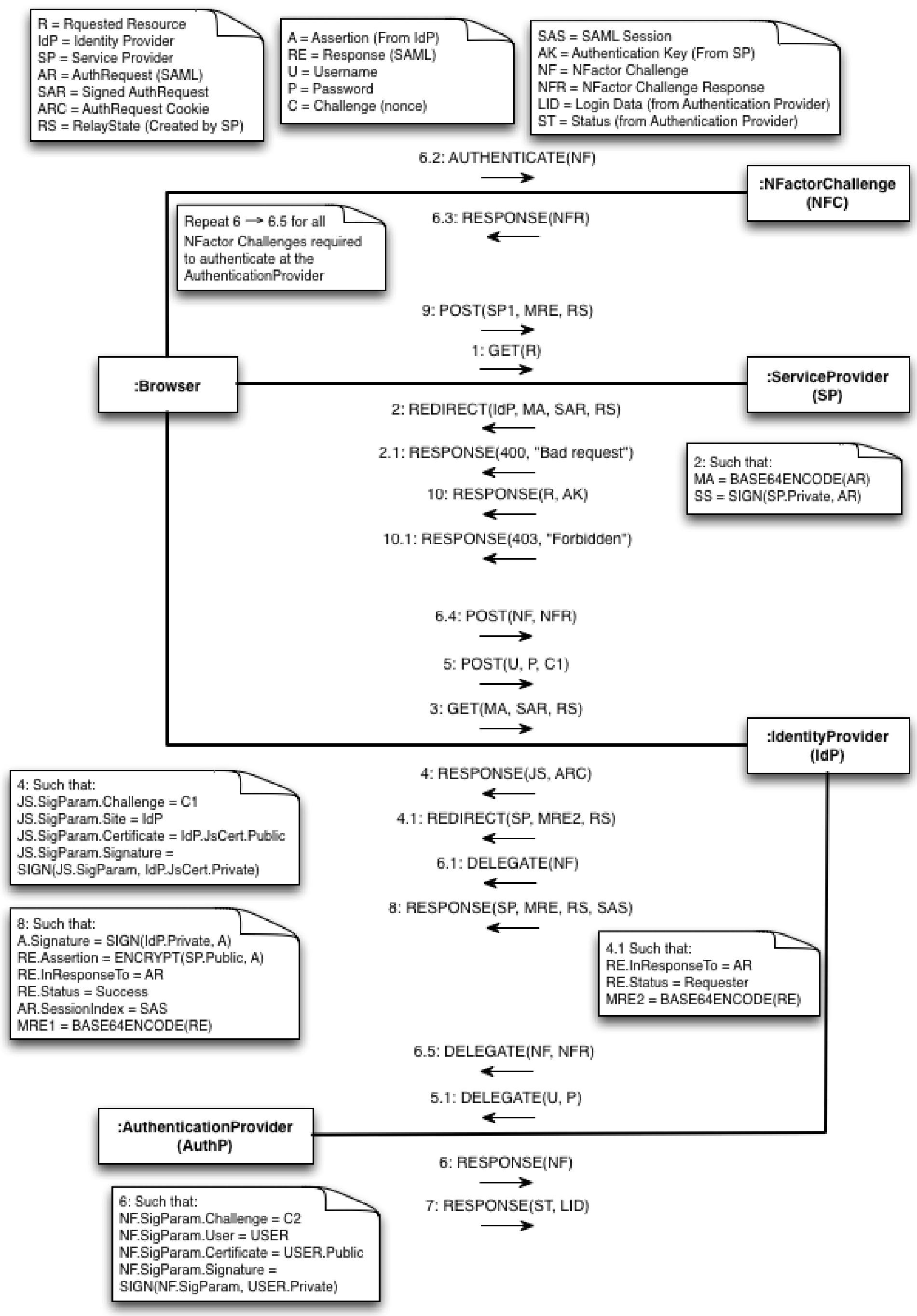

Figure 3.6: Communication diagram for the OpenNemID protocol

Message descriptions

Process 1 describes the creation of the AuthRequest at the service provider when a resource has been requested. If the requested resource is not found a 400 Bad-Request message is returned. The use of url encoding (UrlEnc) and base64 encoding (Base64Enc) is to ensure that the data can be transferred as parameters in the URL.

SAML and OIOSAML has a substantial amount of processing rules that dictates the processing of messages, disobeying these will cause an error. For the sake of communication the descriptions have been kept to a minimum, not listening all scenarios that would cause an error.

Algorithm 1 Process 1

```
Require: GET is well-formed and IdP.Public and SP.Private
  if R exists then
    AR ← CreateAuthnRequest()
    SAR ← SIGN(AR, SP.Private)
    MA ← UrlEnc(Base64Enc(DeflateCompress(AR)))
    RS ← UrlEnc(Base64Enc(R))
    return REDIRECT(IdP, MA, SAR, RS)
  else
    return RESPONSE(400, BadRequest)
  end if
```

Process 3 describes the handling of a *AuthnRequest* at the identity provider. The request will be handled in one of two ways depending on whether the AuthnRequest could be verified. In the case of verification failing, a SAML response will be returned indicating an error. Otherwise a page for requesting the users credentials is returned. The page returned will also contain the signed JavaScript for handling the OpenNemID specific actions. The JavaScript is signed using a certificate, *IdPJsCert*, this certificate has to have been issued by the authentication provider.

A challenge (nonce) is generated, this is to prevent an attack where an adversary would submit an identical request to one that has already been submitted, also known as a *replay attack*. If the challenge wasn't introduced the second identical request could possibly be accepted as well.

A cookie is created containing the AuthnRequest, signed AuthnRequest along with the relay state. This is done to free the identity provider from having to store the data, thereby granting more statelessness.

Algorithm 2 Process 3

```
Require: GET is well-formed and IdP.Private and SP.Public and IdPJsCert.Public and IdP has JavaScript from AuthP
  AR ← DeflateDecompress(Base64Dec(UrlDec(MA)))
  if VERIFY(AR, SAR, SP.Public) then
    C1 ← GenChallenge()
    JS ← GetStoredJavaScript()
    JS.SigParams.Challenge ← C1
    JS.SigParams.Certificate ← IdPJsCert.Public
    JS.SigParams.Signature ← SIGN(JS.SigParams, IdPJsCert.Private)
    ARC ← CreateCookie(MA, SAR, RS)
    return RESPONSE(JS, ARC)
  else
    RE ← CreateResponse()
    RE.InResponseTo ← AR
    RE.Status ← Requester
    MRE ← Base64Enc(RE)
    return REDIRECT(SP, MRE, RS)
  end if
```

As specified in the formalization, it's required for the browser to allow JavaScript. The username, *U* and password, *P* for the user is not stored and fetched directly, but will be input by the user manually, though for the description of process 4 it will be assumed that they are both ready right away. Before the user is prompted for username and password, the JavaScript is verified. The challenge is submitted together with the username and password. Hashing of the username and password has been omitted as it holds no functional enhancement towards the protocol.

Algorithm 3 Process 4

```
Require: U and P and Browser allows JavaScript
  SigParams ← Js.SigParams
  if VERIFY(SigParams, SigParams.Signature, SigParams.Certificate) then
    C1 ← SigParams.Challenge
    return POST(U, P, C1)
  else
    print ERROR
  end if
```

In process 5 the identity provider confirms that the challenge received matches a previously issued challenge, and that is has not already been used. A SAML response indicating an error is returned if the challenge is not accepted, otherwise the username and password will be delegated to the authentication provider.

Algorithm 4 Process 5

```
Require: POST is well formed
  if C1 matches challenge issued by IdP and C1 is valid then
    Delegate U and P to AuthP
  else
    return RESPONSE(ERROR)
  end if
```

Process 5.1 describes how the username and password submitted is used to identify a user at the authentication provider. If a user is found and is valid, a challenge will be generated and the next n-factor challenge for the user will be fetched. The n-factor challenge is signed and then returned. Were no valid user found based on the supplied username and password, a SAML response message indicating an error will be returned.

Algorithm 5 Process 5.1

```
  USER ← GetUser(U, P)
  if USER is valid then
    C2 ← GenChallenge()
    NF ← GetNextNFactorChallenge(USER)
    NF.SigParam.User ← USER
    NF.SigParam.Challenge ← C2
    NF.SigParam.Certificate ← USER.Public
    NF.SigParam.Signature ← SIGN(NF.SigParam, USER.Private)
    return RESPONSE(NF)
  else
    return RESPONSE(ERROR)
  end if
```

Process 6 show how the identity provider has to delegate messages from the browser to the authentication provider and vice versa. In this case the N-Factor challenge is to be delegated. If the authentication provider returned an error or the identity provider cannot verify the n-factor challenge received, a SAML response message indicating an error will be returned to the browser. If the n-factor challenge is verified, it is then delegated to the browser.

Algorithm 6 Process 6

```
  SigParams ← NF.SigParams
  if VERIFY(SigParams, SigParams.Signature, SigParams.Certificate) then
    Delegate NF to Browser
  else
    Delegate ERROR to Browser
  end if
```

The browser also verifies that the n-factor challenge is valid in process 6.1.

An attempt to authenticate the n-factor challenge is then performed. Due to the challenges genericity, the authentication process will not be further described.

Algorithm 7 Process 6.1

```
SigParams ← NF.SigParams
if VERIFY(SigParams, SigParams.Signature, SigParams.Certificate) then
    AUTHENTICATE(NF)
else
    print ERROR
end if
```

In process 6.2, the genericity of the n-factor challenges once more results in a minimal description. The way the authentication attempts will be handled depends on the n-factor challenge, therefore to simplify the description, it is assumed that a result conforming to a specific template is returned.

Algorithm 8 Process 6.2

```
NFR ← NFactorResult(NF)
return RESPONSE(NFR)
```

Process 6.5 describes the handling of a n-factor challenge result. First off, the n-factor challenge is verified to make sure that it has not been altered. Then it is checked whether or not the n-factor challenge result is acceptable for the n-factor challenge. In the case of acceptance the user is fetched. If the user has not yet completed all n-factor challenges required to authenticate at the authentication provider, then the next challenge is fetched, signed and returned to the user. Process 6 to 6.5 is then repeated until no more n-factor challenges are required, in which case login data for the user is created and returned to the identity provider. In case of the n-factor challenge not passing verification or the n-factor challenge result is not accepted a SAML message indicating an error is returned.

Algorithm 9 Process 6.5

```
Require: User identifiable by (NF.SigParams.USER, NF.SigParams.Certificate)
  SigParams ← NF.SigParams
  if VERIFY(SigParams, SigParams.Signature, SigParams.Certificate) then
    if NFR is acceptable result of NF then
      USER ← GetUser(SigParams.USER, SigParams.Certificate)
      C2 ← GenChallenge()
      if USER.HasNextChallenge then
        NF ← GetNextNFactorChallenge(USER)
        NF.SigParams.User ← USER
        NF.SigParams.Challenge ← C2
        NF.SigParams.Certificate ← USER.Public
        NF.SigParams.Signature ← SIGN(NF.SigParams, USER.Private)
        return RESPONSE(NF)
      else
        LID ← CreateLogInData(USER)
        ST ← "OK"
        return RESPONSE(ST, LID)
      end if
    else
      return RESPONSE(ERROR)
    end if
  else
    return RESPONSE(ERROR)
  end if
```

In process 7 it is described how the identity provider handles when a user has been authenticated at the authentication provider. If the returned status does not equal an acceptance criteria, the status is returned to the browser to indicate an error, the status would further describe the error. When the status is accepted the cookie created in process 3 is fetched and the contents of the cookie are extracted.

The initial AuthnRequest extracted from the cookie is verified. Verification failure results in the user being redirected to the service provider with information indicating that the AuthnRequest could not be granted.

When verification succeeds, an assertion is build based on the login data received from the authentication provider. The assertion is then encrypted using the service providers public key. A SAML response is created and the assertion is appended to it along with the AuthnRequest. The browser is then redirected to the service provider.

Algorithm 10 Process 7

```
Require: SP.Public and LID is well-formed and ARC cookie present
  if ST = "OK" then
    MA ← ARC.AR
    SAR ← ARC.SAR
    RS ← ARC.RS
    AR ← DeflateDecompress(Base64Dec(UrlDec(MA)))
    if VERIFY(AR, SAR, SP.Public) then
      A ← BuildAssertion(LID.Certificate)
      SI ← GenerateSessionIndex()
      A.InResponseTo ← AR
      A.Issuer ← IdP
      A.Audience ← SP
      A.SessionIndex ← SI
      A.Signature ← SIGN(A, IdP.Private)
      EA ← ENCRYPT(A, SP.Public)
      RE ← CreateResponse()
      RE.Assertion ← EA
      RE.InResponseTo ← AR
      RE.Status ← "Success"
      MRE ← DeflateCompress(Base64Enc(UrlEnc(RE)))
      SAS ← CreateSAMLSession(SI, SP, LID.CertificateSubject)
      return REDIRECT(SP, MRE, RS, SAS)
    else
      RE ← CreateResponse()
      RE.InResponseTo ← AR
      RE.Status ← "Requester"
      MRE ← DeflateCompress(Base64Enc(UrlEnc(RE)))
      return REDIRECT(SP, MRE, RS)
    end if
  else
    return RESPONSE(ST)
  end if
```

Process 9 describes the last step in the diagram. The SAML response is received at the service provider. The assertion is decrypted and verified. If the verification fails an error message is returned to the browser. If the verification succeeds, an AuthKey is generated. If the user supplies the AuthKey in subsequent requests to the service provider, the service provider will know that the user is already authenticated, thereby eliminating the need to authenticate again until the AuthKey expires. The initial requested resource is retrieved and returned to the browser together with the AuthKey.

Algorithm 11 Process 9

```
Require: POST is well.formed and SP.Private and IdP.Public
  RE ← UrlDec(Base64Dec(DeflateDecompress(MRE)))
  A ← DECRYPT(RE.Assertion, SP.Private)
  if VERIFY(A, A.Signature, IdP.Public) then
    AK ← GenAuthKey()
    R ← Base64Dec(UrlDec(RS))
    RES ← GetResource(R)
    return RESPONSE(RES, AK)
  else
    return RESPONSE(403, Forbidden)
  end if
```

This concludes the formalizing of the OpenNemID protocol and processing, the subsequent chapter will show how to transform this model into an executable modal using F*.

Chapter 4

Modelling with F*

We briefly introduced F* in chapter 2, but will now give a more detailed introduction. F* can be used to model a security protocol. Despite being a formal specification language F* is also executable. F* is described as a *A Verifying Compiler for Distributed Programming*. This chapter will describe how we have used F* to build a formal specification of OpenNemID.

4.1 Introducing F*

F* is a research language from Microsoft Research. F* primarily subsumes two research languages from Microsoft Research, F7[1] and Fine[2]. F* is at this time considered to be an α-release. The purpose of designing F* is to enable the construction and communication of proofs of program properties and of properties of a program's environment in a verifiable secure way. F* is a dialect of ML and compiles to .NET bytecode in a type-preserving style. This means that it can interop with other .NET languages and the types defined in F* can be used by other .NET languages without losing type information. Furthermore there also exists a fully abstract compiler from F* to JavaScript. This makes it possible to deploy F* programs on web pages as JavaScript meanwhile there is a formal guarantee that the program still behaves just as they would according to F* semantics. The compiling and type-checking of F* code utilizes the Z3[3] SMT solver for proving assumptions made with refinement types. F* has been formalized and verified using Coq[4].

[1]http://research.microsoft.com/en-us/projects/f7/
[2]http://research.microsoft.com/en-us/projects/fine/
[3]http://z3.codeplex.com/
[4]Coq is an interactive theorem prover written in OCaml.

4.2 Syntax and semantics

F* inherits syntax and semantics from ML. F* is a functional language which means that it has features like immutability by default, polymorphic types and type inference. In Listing 4.1 we have shown the classic Hello World example in F*. This example shows how to specify a main method by defining a function _ (underscore) at the end of a module. This will instruct the compiler to make an .exe file instead of a dll.

```
module HelloWorld

let _ = print "Hello world!"
```

Listing 4.1: Hello World example in F*

The example in listing **??** shows how to explicitly specify types with the *colon* operator and the *val* declaration for defining function signatures. This example defines the function multiply that takes two integers as parameters and returns an integer. After that it defines the corresponding *let* binding which defines that the multiply method multiplies the 2 arguments. It is important to note that not defining the corresponding let binding will not cause the compilation to fail but give the following warning: *Warning: Admitting value declaration Multiplication.multiply as an axiom without a corresponding definition.* So the value declaration was valid but there is no concrete implementation supporting this claim.

```
module Multiplication

val multiply: x:int -> y: int -> int

let multiply x y = x * y

let mul = multiply 3 4
```

Listing 4.2: Multiplication example in F*

4.3 Refinement types

F* has derived the feature refinement types from the Microsoft Research projects, F7 and Fine. Refinement types are used to make type safe refinements of existing types. Thus it is possible to restrict or refine values more than their original type. Listing 4.3 shows an example with the refinement *nat* of int that states that *nat* will always be zero of larger, i.e. a natural number. The example also shows an attempt to assign -1 to a type of nat which will fail type checking.

```
(*Declare a type nat of natural numbers*)
type nat = i:int{0 <= i}

let x:nat = 1
let y:nat = 0
let z:nat = 1 - 2 (*Will fail type check*)
```

Listing 4.3: Simple refinement types in F*

Refinement types have the form $x : t\{t'\}$ as shown above. So a refinement type is created by taking an existing type and decorating it with an expression in curly brackets. In the example above the refinement type is a simple boolean expression but refinements are not limited to boolean expression. This is extended in F* by its kind-system.

4.3.1 Kinds

Kinds can be seen as an abstraction over types - types can either *have* or be *of* a kind. The *-kind indicates 'regular types' in F*. This covers all the possible types to create in a regular type system for a programming language like Java. Refinement types are of the E-kind and not of the *-kind. The E(rasable)-kinds have no significance at runtime. They only have an effect at the compiling time during type checking.

4.4 Understanding F*

Since we are extending Jacob's work with the authentication part of the protocol we used his code as a reference for implementing the rest of the OpenNemID protocol. In listing 4.4 we show Jacob's implementation of the Identity Provider. He has implemented a recursive function *identityprovider* declared with the *val* binding just above it. The function declared takes 2 arguments and returns *unit*, which is the same as void in Java and C#.

The arguments

1. a principal for identifying the identity provider
2. a principal for identifying the client.

```
module Identityprovider

open SamlProtocol
open Crypto

let handleUserAuthenticated me user client authnrequest =
  match authnrequest with
  | MkAuthnRequest(reqid,issueinst,dest,sp,msg,sigSP) ->
      let pubksp = CertStore.GetPublicKey sp in

      if (VerifySignature sp pubksp msg sigSP) then
        (assert (Log sp msg);
        let assertion = IssueAssertion me user sp reqid in
        let myprivk = CertStore.GetPrivateKey me in
        assume(Log me assertion);
        let sigAs = Sign me myprivk assertion in
        let signAssertion = AddSignatureToAssertion assertion
            sigAs in
        let encryptedAssertion = EncryptAssertion sp pubksp
signAssertion in
        let resp = AuthResponseMessage me sp encryptedAssertion
             in
        SendSaml client resp) (*10*)
      else
        SendSaml client (Failed Requester)(*10.2*)

 val identityprovider: me:prin -> client:prin -> unit

 let rec identityprovider me client =
  let req = ReceiveSaml client in (*3 & 11*)
  match req with
  | AuthnRequestMessage (issuer, destination, message, sigSP) ->
     let pubkissuer = CertStore.GetPublicKey issuer in
     if (VerifySignature issuer pubkissuer message sigSP) then
       (assert (Log issuer message);
       let challenge = GenerateNonce me in
       let resp = UserCredRequest challenge in
       SendSaml client resp; (*4*)
       identityprovider me client (*Start over*))
     else
       SendSaml client (Failed Requester);(*4.1*)
       identityprovider me client (*Start over*)
   | UserAuthenticated (status, logindata, authnrequest) ->

     match logindata with
     | MkLoginData (user,sig,cert,challenge,site,data) ->
       if (status = "OK") && (VerifySignature user cert data sig
           ) then
         (assert (Log user data);
           handleUserAuthenticated me user client authnrequest;
           identityprovider me client (*Start over*)
         )
       else
         SendSaml client (DisplayError 400);(*10.1*)
         identityprovider me client (*Start over*)
   | _ -> SendSaml client (DisplayError 400);(*10.1*)
         identityprovider me client (*Start over*)
```

Listing 4.4: NemID identity provider implementation [1]

The recursive function *identityprovider* starts by receiving a SAML message from the client. It then matches the request with a *SamlMessage.AuthnRequestMessage* or*SamlMessage.UserAuthenticated* type. When matched with an AuthnRequest message it verifies the Service Providers signature of the message by the function *VerifySignature* which is shown in listing 4.5. The function takes a principal, the principals public key, a message and a signature. It returns a boolean indicating if the check passed. The return type however has a refinement type that relates the message to the principal if the verification passes. ==> should be seen as an implication therefore stating that the predicate is valid. If the verification of the Service Providers signature of the AuthnRequestMessage passes it creates a nonce to be related to this user when the user has authenticated himself/herself by NemID and sends the response to the user. When the user has authenticated through NemID the function *handleUserAuthenticated* is called. The functions purpose is to issue a signed assertion and sending the AuthResponseMessage to the user. The signature for signing messages takes 4 arguments - the principal, the signer, the private key of the principal and the message to be signed. The message is annotated with a refinement type {*Log p msg*}. This refinement type is an E-kinded type that takes a principal and a string as constructor elements. The val declaration for a method Sign in listing 4.5 requires the predicate {Log p msg} to be true before it can typecheck. This means that the message to sign is related to the principal signing the message. Securing this is done by calling *assume*{*Log me assertion*} before signing the message. The predicate is by virtue of this "verified". After this the assertion is encrypted by using the Service Providers public key and sent within an AuthResponseMessage to the user.

```
type pubkey :: prin => *
type privkey :: prin => *

type Log :: prin => string => E

val Sign: p:prin
 -> privkey p
 -> msg:string{Log p msg}
 -> dsig

val VerifySignature: p:prin
 -> pubkey p
 -> msg:string
 -> dsig
 -> b:bool{b=true ==> Log p msg}
```

Listing 4.5: Cryptographic elements

In listing 4.5 we show the declaration of the types for private key (*privkey*) and public key (*pubkey*). These types are declared by using the F* syntax for constructing dependent types (the double colon). This means that a type *pubkey* will have a constructor that takes a *prin* (principal) and returns a type of *-kind. This is still abstract and the type has no actual constructor.

4.5 OpenNemID specified in F*

The code in this section represents the state of the project now. This is in no way a complete implementation of the protocol. Implementation was carried out in an incremental manner. First the focus was on understanding Jacob's work and expanding that with the authentication part (Authentication Provider) of the protocol, which before was done by NemID, and then adding the functionality of creating the login, establishing a connection between Identity Provider and the Authentication Provider and so on. All source code that has been produced in this project can be found on the source code sharing community Github[5]. The F* code for the protocol is organized in 10 modules:

1. The TypeFunc module
2. The SamlProtocol module
3. The Crypto module
4. The CertStore module
5. The Messaging module
6. The Service Provider module
7. The Identity Provider module
8. The Database module
9. The Authentication Provider module
10. The Browser module

The modeling follows the principles for cryptographic protocol modeling outlined by Dolev & Yao[6]. In the following we will explain the important principles for each module and the relation to the algorithms outlined in chapter 3.

4.5.1 Specification of the type functionality module

```
module TypeFunc

type Authentication =
  | Facebook: id:int -> Authentication
  | SMS: generated:int -> Authentication
 | Google: id:int -> Authentication
 | OpenId: id:int -> Authentication
```

Listing 4.6: TypeFunc module

The *TypeFunc* module provides the type authentication which is used for the different kinds of n factor authentication. Note that currently there is only an id associate with each type of authentication for simplicity. This needs to be modified so that each type is more explicit and holds the correct information for authentication.

[5]https://github.com/kiniry-supervision/OpenNemID

[6]Cryptographic primitives are assumed perfect and cyphers cannot be decrypted without the the proper decryption key

4.5.2 Specification of the SAML Protocol

```
module SamlProtocol

open Crypto
open TypeFunc

type assertiontoken = string (*Add refinements*)
type signedtoken = string (*Add refinements*)
type id = string
type endpoint = string
type uri = string


type AuthnRequest =
  | MkAuthnRequest: IssueInstant:string ->
      Destination:endpoint -> Issuer:prin ->
      message:string -> sig:dsig ->
      AuthnRequest

type LoginData =
  | MkLoginData:  user:prin -> signature:dsig ->
    cert:pubkey user -> challenge:nonce ->
    site:string -> data:string ->
    LoginData

type LoginInfo =
  | UserLogin:  userid:string -> password:string ->
  LoginInfo

type AuthInfo =
  | UserAuth:   userid:string -> authmethod:Authentication ->
  authresponse:Authentication -> AuthInfo

type Assertion =
  | SignedAssertion: assertiontoken -> dsig -> Assertion
  | EncryptedAssertion: cypher -> Assertion

type SamlStatus =
  | Success: SamlStatus
  | Requester: SamlStatus
  | Responder: SamlStatus

type LoginError =
  | AuthError: LoginError
  | CredentialError: LoginError

type SamlMessage =
  | SPLogin: uri -> SamlMessage
  | Login: loginInfo:LoginInfo -> challenge:nonce ->
      SamlMessage
  | LoginResponse: string -> SamlMessage
  | AuthnRequestMessage: issuer:prin ->  destination:endpoint
      -> message:string -> dsig -> SamlMessage
  | LoginRequestMessage: issuer:prin ->  destination:endpoint
      -> loginInfo:LoginInfo -> SamlMessage
    | NfactAuthRequest: issuer:prin -> destination:endpoint ->
      authInfo:AuthInfo -> challenge:nonce -> dsig ->
      SamlMessage
    | AuthResponseMessage: issuer:prin -> destination:endpoint ->
       Assertion -> SamlMessage
    | LoginResponseMessage: issuer:prin -> destination:endpoint
      -> auth:Authentication -> challenge:nonce -> dsig ->
      SamlMessage
    | UserAuthenticated: status:string -> logindata:LoginData ->
      authnReq:AuthnRequest -> SamlMessage
    | UserCredRequest: javascript:string -> challenge:nonce ->
      dsig -> SamlMessage
    | UserAuthRequest: authmethod:Authentication -> challenge:
      nonce -> dsig -> SamlMessage
    | UserAuthResponse: authInfo:AuthInfo -> challenge:nonce ->
      dsig -> SamlMessage
    | LoginSuccess: status:string -> issuer:prin -> destination:
      endpoint -> SamlMessage
    | Failed: SamlStatus -> SamlMessage
    | LoginFailure: LoginError -> SamlMessage
    | DisplayError: int -> SamlMessage


val SendSaml: prin -> SamlMessage -> unit
val ReceiveSaml: prin -> SamlMessage

val CreateAuthnRequestMessage: issuer:prin -> destination:prin
    -> string
val IssueAssertion: issuer:prin -> subject:prin -> audience:
    prin -> inresto:AuthnRequest -> assertiontoken
val AddSignatureToAssertion: assertiontoken -> dsig ->
    signedtoken
val EncryptAssertion: receiver:prin -> pubkey receiver ->
    signedtoken -> Assertion
val DecryptAssertion: receiver:prin -> privkey receiver ->
    Assertion -> (signedtoken * dsig)
```

Listing 4.7: Specification of the SAML Protocol elements

The *SamlProtocol* module is taken directly from Jacob's code and only modified to support more and different *SamlMessage* that are needed in our specification of OpenNemID. This module's purpose is the specification of messages and to provice functions for sending and receiving messages. Note that the functions for sending and receiving messages have no runtime implementation. They are only specified by the *val* declaration. The SAML Protocol is used for the communication between the principals in the OpenNemID protocol in a login session. The intention with these functions is that they will handle the mapping of protocol elements to the network.

4.5.3 Specification of cryptographic elements

```
module Crypto

open Protocol
open TypeFunc

type prin = string
type pubkey :: prin => *
type privkey :: prin => *
type dsig
type nonce = string
type cypher

(*Verification*)
type Log :: prin => string => E

type LogAuth :: prin => Authentication => E

val Keygen: p:prin
   -> (pubkey p * privkey p)

val Sign: p:prin
 -> privkey p
 -> msg:string{Log p msg}
 -> dsig

val SignAuth: p:prin
 -> privkey p
 -> msg:Authentication{LogAuth p msg}
 -> dsig

val VerifySignature: p:prin
 -> pubkey p
 -> msg:string
 -> dsig
 -> b:bool{b=true ==> Log p msg}

val VerifySignatureAuth: p:prin
 -> pubkey p
 -> msg:Authentication
 -> dsig
 -> b:bool{b=true ==> LogAuth p msg}

val Encrypt: p:prin
 -> pubkey p
 -> string
 -> cypher

val Decrypt: p:prin
 -> privkey p
 -> cypher
 -> string

val GenerateNonce: prin -> nonce (*Add refinement to ensure
    unqueness*)
```

Listing 4.8: Specification of cryptographic elements

The *crypto* module is taken directly from Jacob's code and only modified to support signing and verification of the authentication type. The purpose of the *crypto* is providing the cryptographic functions to sign and verify both messages and the authentication type also the encryption and decryption of messages. The *crypto* module utilizes the refinement type to ensure that signed messages and authentications have typed dependency to the signing principal. It does not have a concrete implementation as of now.

4.5.4 Specification of certificate store module

```
module CertStore

open Crypto

val GetPublicKey: p:prin -> pubkey p
val GetJSPublicKey: p:prin -> pubkey p
(*Prin needs to be updated to include credentials*)
val GetPrivateKey: p:prin -> privkey p
val GetJSPrivateKey: p:prin -> privkey p
```

Listing 4.9: Abstract certificate store

The *CertStore* module is taken from Jacob's code and expanded with functionality to support JavaScript public and private keys. This module provides four abstract functions for retrieving certificates from a certificate store. The functions use the value dependent syntax for relating a principal to the certificate keys. As Jacob has written in a comment the principal could be updated to include credentials because this is a quite naive implementation. It is quite naive because all you need to obtain the private key of a principal is the name of the principal.

4.5.5 Specification of the messaging protocol

```
module Messaging

open Crypto
open TypeFunc

type Status =
 | Successful: Status
 | Unsuccessful: Status

type Message =
 | NewSiteRequest: idp:prin -> Message
 | ChallengeResponse: challenge:nonce -> Message
 | IdpChalResponse: challenge:nonce -> Message
 | AcceptedIdp: idp:prin -> pubkey:pubkey idp -> authp:prin ->
     authpubkey:pubkey authp -> signedjavascript:string ->
     Message
 | RequestForLogin: passportnumber:string -> Message
 | ReqLoginResponse: challenge:nonce -> Message
 | CreateLogin: generatedpassword:string -> challenge:nonce ->
     Message
 | ChangeUserId: userid:string -> newUserId:string -> password:
     string -> Message
```

```
  | ChangePassword: userid:string -> password:string ->
      newPassword:string -> Message
  | UserRevokeIdp: userid:string -> password:string -> idp:
      string -> Message
  | AddNfactor: userid:string -> password:string -> nfact:
      Authentication -> Message
  | RemoveNfactor: userid:string -> password:string -> nfact:
      Authentication -> Message
  | StatusMessage: Status -> Message


val SendMessage: prin -> Message -> unit
val ReceiveMessage: prin -> Message
```

Listing 4.10: Specification of the Messaging protocol

The *Messaging* module is responsible for 2 things - the specification of messages and providing functions for sending and receiving these messages. As the *Saml-Protocol* module the functions for sending and receiving are specified only by the *val* declaration and has no concrete runtime implementation. This module is used to model the communication between Identity Provider / user and the Authentication Provider when wanting to establish a secure connection and creating and/or changing an user's login.

4.5.6 Specification of the Service Provider

```
module Serviceprovider

open SamlProtocol
open Crypto

val serviceprovider:  me:prin -> client:prin -> idp:prin ->
    unit

let rec serviceprovider me client idp =
 let req = ReceiveSaml client in
 match req with
  | SPLogin (url) ->
    let authnReq = CreateAuthnRequestMessage me idp in
    assume(Log me authnReq);
    let myprivk = CertStore.GetPrivateKey me in
    let sigSP = Sign me myprivk authnReq in
    let resp = AuthnRequestMessage me idp authnReq sigSP in
    SendSaml client resp;
    serviceprovider me client idp
  | AuthResponseMessage (issuer, destination, encassertion) ->
    let myprivk = CertStore.GetPrivateKey me in
    let assertion = DecryptAssertion me myprivk encassertion in
    match assertion with
    | SignedAssertion (token,sigIDP) ->
      let pubkissuer = CertStore.GetPublicKey idp in
      if VerifySignature idp pubkissuer token sigIDP
      then
        (assert(Log idp token);
        let resp = LoginResponse "You are now logged in" in
```

```
          SendSaml client resp)
        else SendSaml client (DisplayError 403);
        serviceprovider me client idp

    | _ -> SendSaml client (DisplayError 400);
          serviceprovider me client idp
```

Listing 4.11: Specification of service provider

The service provider is taken and directly from Jacob's code and it is not modified in any way. The service provider implements algorithm 1 and 11 in section 3.4. The module is constructed to accept SAML messages of type *SPLogin* and *AuthResponseMessage*. If the service provider receives another type of message it will return a HTTP error. Contrary to algorithms 1 and 11 the service provider does not implement encoding and decoding because this is expected to be handled by the *SamlProtocol* module.

4.5.7 Specification of the Identity Provider

The specification of the Identity Provider is divided into several listings for the sake of understandability.

```
module Identityprovider

open SamlProtocol
open Crypto
open TypeFunc
open Messaging

val userloggedin: user:prin -> bool
val getjavascript: string
val userlogin: user:prin -> unit
val decodeMessage: message:string -> AuthnRequest
val getauthnrequest: user:prin -> challenge:nonce ->
    AuthnRequest
val getuserchallenge: user:prin -> nonce
val relatechallenge: user:prin -> challenge:nonce -> unit
val verifychallenge: user:prin -> challenge:nonce -> bool
val relate: user:prin -> challenge:nonce -> authnReq:
    AuthnRequest -> unit

val identityprovider: me:prin -> user:prin -> authp:prin ->
    unit

let rec identityprovider me user authp =
 let request = ReceiveSaml user in
 match request with
 | AuthnRequestMessage(issuer, destination, message, sigSP) ->
  let pubkissuer = CertStore.GetPublicKey issuer in
 if (VerifySignature issuer pubkissuer message sigSP) then
  (assert (Log issuer message);
  let authnReq = decodeMessage message in
  let myprivk = CertStore.GetPrivateKey me in
  if not (userloggedin user) then
   let challenge = GenerateNonce me in
```

```
    relate user challenge authnReq;
    relatechallenge user challenge;
    let js = getjavascript in
    assume(Log me js);
    let myprivk = CertStore.GetJSPrivateKey me in
    let sigIdP = Sign me myprivk js in
    let resp = UserCredRequest js challenge sigIdP in
    SendSaml user resp;
    identityprovider me user authp
   else
    let assertion = IssueAssertion me user issuer authnReq in
    assume(Log me assertion);
    let myprivk = CertStore.GetPrivateKey me in
    let pubksp = CertStore.GetPublicKey issuer in
    let sigAs = Sign me myprivk assertion in
    let signAssertion = AddSignatureToAssertion assertion sigAs
        in
    let encryptedAssertion = EncryptAssertion issuer pubksp
        signAssertion in
    let resp = AuthResponseMessage me issuer encryptedAssertion
        in
    SendSaml user resp)
  else
   SendSaml user (Failed Requester);
   identityprovider me user authp
  | Login (loginInfo, challenge) ->
   if (verifychallenge user challenge) then
    let req = LoginRequestMessage me authp loginInfo in
    SendSaml authp req;
    handleauthresponse me user authp;
    identityprovider me user authp
   else
    SendSaml user (DisplayError 400);
    identityprovider me user authp
  | UserAuthResponse(authInfo, challenge, sigAuth) ->
   let req = NfactAuthRequest me authp authInfo challenge
       sigAuth in
   SendSaml authp req;
   handleauthresponse me user authp;
   identityprovider me user authp
  | _ -> SendSaml user (DisplayError 400);
   identityprovider me user authp
```

Listing 4.12: Handling and delegation of a user's requests

This part of the identity provider implements the algorithms 2 and . The identity provider accepts the three SAML messages *AuthnRequestMessage*, *Login* and *UserAuthResponse* from the user.

1. The *AuthnRequestMessage* branch decodes the message and if the user has not logged in previously it sends a *UserCredRequest* back with the JavaScript and a nonce to be used for relating the login at the Identity Provider prompting the user to give his or her login information. If the user has already logged in previously it issues an assertion to the user.

2. The *Login* branch handles the user's login information which is the re-

sponse the user provides after receiving the *UserCredRequest*. This branch verifies that the nonce from the user is the correct one and if it is correct it delegates the login information to the Authentication Provider and then calls the function *handleauthresponse* which we will explain later in this section. If the nonce is incorrect it returns a HTTP error.

3. The *UserAuthResponse* branch handles the user's n factor authentication information and delegates the information to the Authentication Provider.

```
val handleUserAuthenticated: me:prin -> user:prin -> authnReq:
    AuthnRequest -> unit

let handleUserAuthenticated me user authnReq =
 match authnReq with
 | MkAuthnRequest(issueinst,dest,sp,msg,sigSP) ->
  let pubksp = CertStore.GetPublicKey sp in
   if (VerifySignature sp pubksp msg sigSP) then
  (assert (Log sp msg);
  let assertion = IssueAssertion me user sp authnReq in
  let myprivk = CertStore.GetPrivateKey me in
  assume(Log me assertion);
  userlogin user;
  let sigAs = Sign me myprivk assertion in
  let signAssertion = AddSignatureToAssertion assertion sigAs
      in
  let encryptedAssertion = EncryptAssertion sp pubksp
      signAssertion in
  let resp = AuthResponseMessage me sp encryptedAssertion in
  SendSaml user resp)
      else
 SendSaml user (Failed Requester)

val handleauthresponse: me:prin -> user:prin -> authp:prin ->
    unit

let handleauthresponse me user authp =
 let resp = ReceiveSaml authp in
 match resp with
 | LoginResponseMessage(issuer, destination, authmethod,
     challenge, sigUser) ->
  let pubkeyuser = CertStore.GetPublicKey user in
  if VerifySignatureAuth user pubkeyuser authmethod sigUser
      then
   (assert (LogAuth user authmethod);
   relatechallenge user challenge;
   let resp = UserAuthRequest authmethod challenge sigUser in
   SendSaml user resp)
  else
   SendSaml user (DisplayError 403)
 | LoginSuccess(status, issuer, destination) ->
  if (status = "OK") then
   let challenge = getuserchallenge user in
   let authnReq = getauthnrequest user challenge in
   handleUserAuthenticated me user authnReq
  else
   SendSaml user (DisplayError 403)
  | _ -> SendSaml user (DisplayError 400)
```

Listing 4.13: The handling of the responses from Authentication Provider

This part of the identity provider handles the information received from the Authentication Provider. It implements algorithms 6 and 7. The function *handleauthresponse* has two match branches:

1. *LoginResponseMessage* which will prompt the user for a n factor authentication method while it relates the challenge generated by the Authentication Provider to the user for verification

2. *LoginSuccess* which specifies that the user has passed all the n factor authentication methods.

If the user has been successfully logged in the user will be issued an assertion which is done in the *handleUserAuthenticated* function. This function will also save a cookie that specifies that this user has logged in which the Identity Provider will search for when getting an *AuthnRequestMessage* from a user.

```
val savejavascript: javascript:string -> unit
val savepublickey: owner:prin -> publickey:pubkey owner -> unit

val connectwithauthp: me:prin -> authp:prin -> unit

let connectwithauthp me authp =
 let req = NewSiteRequest me in
 let _ = SendMessage authp req in
 let resp = ReceiveMessage authp in
 match resp with
 | ChallengeResponse(challenge) ->
  let _ = SendMessage authp (IdpChalResponse challenge) in
  let res = ReceiveMessage authp in
  match res with
  | AcceptedIdp(idp, idppubkey, authp, authppubkey, signedjs)
     ->
   (*Established secure connection*)
   savejavascript signedjs;
   savepublickey authp authppubkey;
   savepublickey idp idppubkey
  | _ -> res; ()
 | _ -> resp; ()
```

Listing 4.14: Establising a secure connection with Authentication Provider

This part of the Identity Provider is the establishing of the secure connection between the Identity Provider and the Authentication Provider. Right now the challenge response from the Authentication Provider is just a nonce to illustrate that the Identity Provider needs to be investigated thoroughly by the Authentication Provider for the purpose of finding out if it is a non-evil Identity Provider.

4.5.8 Specification of the Database Handler

```
module Database

open Crypto
open CertStore
open TypeFunc

(*Identity provider functionality*)
val whitelist: idp:prin -> unit
val blacklist: idp:prin -> unit
val addidp: idp:prin -> bool
val whitelisted: idp:prin -> bool

(*User functionality*)
val createuser: user:prin -> userid:string -> password:string
    -> bool
val usercreation: user:prin -> generatedPassword:string -> bool
val changeuserid: user:string -> newuser:string -> password:
    string -> bool
val changeuserpassword: user:string -> password:string ->
    newpassword:string -> bool

val addnfactor: user:string -> password:string -> nfactor:
    Authentication -> bool
val removenfactor: user:string -> password:string -> nfactor:
    Authentication -> bool

val getnfactor: user:string -> Authentication
val checknfactor: user:string -> Authentication -> bool
val allnfactauthed: user:string -> bool
val resetnfact: user:string -> unit

val checklogin: user:string -> password:string -> bool

val revokeidp: user:string -> password:string -> idp:string ->
    bool

val revokedidp: user:string -> idp:prin -> bool
```

Listing 4.15: Specification of the database

The *Database* module is responsible for the communication with the database and therefore checking the information provided by the user. The database is also responsible for keeping track of how many n factor authentications the user has gone through. Note that as of now these functions are just specified by the *val* declaration and therefore has no concrete implementation.

4.5.9 Specification of the Authentication Provider

The specification of the Authentication Provider is divided into several listings for the sake of understandability.

```
module Authenticationprovider

```

```
open SamlProtocol
open Crypto
open Database
open TypeFunc
open Messaging

val relatechallenge: user:prin -> challenge:nonce -> unit

val verifychallenge: user:prin -> challenge:nonce -> bool

val nfactauth: me:prin -> idp:prin -> user:prin -> userid:
    string -> unit

let nfactauth me idp user userid =
 if (allnfactauthed userid) then
  resetnfact userid;
  let status = "OK" in
  let resp = LoginSuccess status me idp in
  SendSaml idp resp
 else
  let challenge = GenerateNonce me in
  let authmethod = getnfactor userid in
  assume(LogAuth user authmethod);
  let userprivkey = CertStore.GetPrivateKey user in
  let sigUser = SignAuth user userprivkey authmethod in
  let resp = LoginResponseMessage me idp authmethod challenge
      sigUser in
  SendSaml idp resp

val authenticationprovider: me:prin -> idp:prin -> user:prin ->
     unit

let rec authenticationprovider me idp user =
 let req = ReceiveSaml idp in
 match req with
 | LoginRequestMessage (issuer, destination, loginInfo) ->
  if (whitelisted idp) then
   match loginInfo with
   | UserLogin(userid, password) ->
    if not (revokedidp userid idp) && (checklogin userid
        password) then
     let challenge = GenerateNonce me in
     let authmethod = getnfactor userid in
     assume(LogAuth user authmethod);
     let userprivkey = CertStore.GetPrivateKey user in
     let sigUser = SignAuth user userprivkey authmethod in
     relatechallenge user challenge;
     let resp = LoginResponseMessage me idp authmethod
         challenge sigUser in
     SendSaml idp resp;
     authenticationprovider me idp user
    else
     SendSaml idp (LoginFailure CredentialError);
     authenticationprovider me idp user
   | _ -> SendSaml idp (Failed Requester);
    authenticationprovider me idp user
  else
```

```
    SendSaml idp (Failed Requester);
    authenticationprovider me idp user
  | NfactAuthRequest(issuer, destination, authInfo, challenge,
      sigAuth) ->
   if (whitelisted idp) then
    match authInfo with
    | UserAuth(userid, authmethod, authresponse) ->
     let userpubkey = CertStore.GetPublicKey user in
     if VerifySignatureAuth user userpubkey authmethod sigAuth
        && verifychallenge user challenge then
      if not (revokedidp userid idp) && (checknfactor userid
          authresponse) then
       nfactauth me idp user userid;
       authenticationprovider me idp user
      else
       SendSaml idp (LoginFailure AuthError);
       authenticationprovider me idp user
     else
      SendSaml idp (LoginFailure AuthError);
      authenticationprovider me idp user
    | _ -> SendSaml idp (Failed Requester);
     authenticationprovider me idp user
   else
    SendSaml idp (Failed Requester);
    authenticationprovider me idp user
  | _ -> SendSaml idp (Failed Requester);
   authenticationprovider me idp user
```

Listing 4.16: Specification of the authentication provider

The Authentication Provider implements algorithms 5 and 9. The Authentication Provider accepts two SAML messages *LoginRequestMessage* and *NfactAuthRequest* from the Identity Provider.

1. The *LoginRequestMessage* branch will check the login information. If the correct login information has been provided by the user it generates a nonce to be related to this user and specifies which type of n factor authentication the user has to go through and that will be sent to the Identity Provider.

2. The *NfactAuthRequest* branch is the receiving of n factor authentication response from the user. It verifies that the sender of the message is the correct user and the n factor information. The correct n factor information will make the function *nfactauth* which will handle if the user has gone through all n factor authentication method or needs to specify more. The function will the information to the Identity Provider.

```
val usercommunication: me:prin -> user:prin -> unit

let rec usercommunication me user =
 let req = ReceiveMessage user in
 match req with
 | RequestForLogin(passportnumber) ->
  if createuser user passportnumber then
```

```
    let challenge = GenerateNonce me in
    relatechallenge user challenge;
    SendMessage user (ReqLoginResponse challenge);
    usercommunication me user
   else
    SendMessage user (StatusMessage Unsuccessful);
    usercommunication me user
  | CreateLogin(generatedpassword, challenge) ->
   if (verifychallenge user challenge) && (usercreation user
       generatedpassword) then
    let challenge = GenerateNonce me in
    relatechallenge user challenge;
    SendMessage user (StatusMessage Successful);
    usercommunication me user
   else
    SendMessage user (StatusMessage Unsuccessful);
    usercommunication me user
  | ChangePassword(userid, password, newPassword) ->
   if changeuserpassword userid password newPassword then
    SendMessage user (StatusMessage Successful);
    usercommunication me user
   else
    SendMessage user (StatusMessage Unsuccessful);
    usercommunication me user
  | ChangeUserId(userid, newUserId, password) ->
   if changeuserid userid newUserId password then
    SendMessage user (StatusMessage Successful);
    usercommunication me user
   else
    SendMessage user (StatusMessage Unsuccessful);
    usercommunication me user
  | UserRevokeIdp(userid, password, idp) ->
   if revokeidp userid password idp then
    SendMessage user (StatusMessage Successful);
    usercommunication me user
   else
    SendMessage user (StatusMessage Unsuccessful);
    usercommunication me user
  | AddNfactor(userid, password, nfact) ->
   if addnfactor userid password nfact then
    SendMessage user (StatusMessage Successful);
    usercommunication me user
   else
    SendMessage user (StatusMessage Unsuccessful);
    usercommunication me user
  | RemoveNfactor(userid, password, nfact) ->
   if removenfactor userid password nfact then
    SendMessage user (StatusMessage Successful);
    usercommunication me user
   else
    SendMessage user (StatusMessage Unsuccessful);
    usercommunication me user
  | _ -> SendMessage user (StatusMessage Unsuccessful);
   usercommunication me user
```

Listing 4.17: The creation and changing of a user's account

This part of the Authentication Provider is responsible for creating and changing a user's account. It is pretty intuitive what the different messages does. When creating a login the user will give an email account where they will receive an email with a one-time password to verify their account. The database will handle all the information and the checking of the information. We wrote in section 2.4 about *N factor authentication* and that a user must have at least one method for n factor authentication. This is not yet enforced by the Authentication Provider when a user registers an account.

```
val getsignedjavascript: string

val establishidp: me:prin -> idp:prin -> unit

let rec establishidp me idp =
 let req = ReceiveMessage idp in
 match req with
 | NewSiteRequest(idp) ->
  let challenge = GenerateNonce me in
  relatechallenge idp challenge;
  SendMessage idp (ChallengeResponse challenge);
  establishidp me idp
 | IdpChalResponse(challenge) ->
  if (verifychallenge idp challenge) && (addidp idp) then
   let idppubkey = CertStore.GetPublicKey idp in
   let mypubk = CertStore.GetPublicKey me in
   let signedjs = getsignedjavascript in
   let resp = AcceptedIdp idp idppubkey me mypubk signedjs in
   SendMessage idp resp;
   establishidp me idp
  else
   SendMessage idp (StatusMessage Unsuccessful);
   establishidp me idp
 | _ -> SendMessage idp (StatusMessage Unsuccessful);
  establishidp me idp
```

Listing 4.18: Established a secure connection with the Identity Provider

This part of the Authentication Provider will handle the establishing of a secure connection between the Identity Provider and the Authentication Provider. As we mentioned when we described the Identity Provider's specification of this model there needs to be some investigation of the Identity Provider and not just a generated nonce. This is just specified to give an idea of how the model is designed.

4.5.10 Specification of the Browser

The specification of the Browser is divided into two listings for the sake of understandability. Note that we can not model the user's input therefore the input from the user is specified by a bunch of *val* declarations.

```
module Browser

open SamlProtocol
open Crypto
open CertStore
open TypeFunc
open Messaging

val loginWithFb: Authentication
val loginWithYubico: Authentication
val loginWithSMS: Authentication
val loginWithOpenId: Authentication
val userid: string
val password: string
val fakeprint: str:string -> unit

val handleAuthMethod: auth:Authentication -> Authentication

let handleAuthMethod auth =
 match auth with
  | Facebook(fbid) -> loginWithFb
  | SMS(gen) -> loginWithSMS
  | OpenId(oid) -> loginWithOpenId
  | Yubico(yid) -> loginWithYubico

val loop: user:string -> idp:prin -> sp:prin -> unit

let rec loop userid idp sp =
 let loginresp = ReceiveSaml idp in
  match loginresp with
  | UserAuthRequest(authmethod, challenge, sigAuth) ->
    let authresponse = handleAuthMethod authmethod in
    let authInfo = UserAuth userid authmethod authresponse in
    let authresp = UserAuthResponse authInfo challenge sigAuth
        in
    SendSaml idp authresp;
    loop userid idp sp
  | AuthResponseMessage(idenp, dest, assertion) ->
    SendSaml sp loginresp
  | _ -> loginresp; ()

val browser: sp:prin -> res:uri -> unit

let browser sp resource =
 let req = SPLogin resource in
 let _ = SendSaml sp req in
  let res = ReceiveSaml sp in
  match res with
  | AuthnRequestMessage(sp, idp, message, sigSP) ->
    let _ = SendSaml idp res in
    let idpResp = ReceiveSaml idp in
    match idpResp with
    | UserCredRequest(javascript, challenge, sigIdP) ->
     let pubkissuer = CertStore.GetJSPublicKey idp in
     if VerifySignature idp pubkissuer javascript sigIdP then
      (assert (Log idp javascript);
      let loginInfo = UserLogin userid password in
      let loginreq = Login loginInfo challenge in
      SendSaml idp loginreq;
      loop userid idp sp;
      let spResp = ReceiveSaml sp in
      match spResp with
      | LoginResponse(str) ->
        fakeprint str
      | _ -> spResp; ())
     else
      fakeprint "Validation Error"
    | _ -> idpResp; ()
   | _ -> res; ()
```

Listing 4.19: Browser's side of logging in

This part of the *Browser* module is used to model the client's side of a logging in session. It implements algorithms 3 and 7. It is worth noticing that the *Browser* verifies the JavaScript is actually received from the correct Identity Provider. The function *fakeprint* is used to give the user messages about errors and if they are logged in. The recursive function *loop* will provide n factor authentication methods until the client receives a *AuthResponseMessage* which it then will send to the Service Provider and the user is now logged in.

```
val newUserId: string
val newPassword: string
val idpToRevoke:string
val nfactToRemove: Authentication
val nfactToAdd: Authentication

val retrieveGeneratedPassword: string

val createUser: authp:prin -> passportnumber:string -> unit

let createUser authp passportnumber =m
 let req = RequestForLogin passportnumber in
 let _ = SendMessage authp req in
  let resp = ReceiveMessage authp in
  match resp with
  | ReqLoginResponse(challenge) ->
   let reqlresp = CreateLogin retrieveGeneratedPassword
       challenge in
   let _ = SendMessage authp reqlresp in
   let createloginresp = ReceiveMessage authp in
   match createloginresp with
   | StatusMessage(status) ->
    match status with
    | Successful -> fakeprint "You have created an account"
    | Unsuccessful -> fakeprint "Something went wrong. No
        account has been created"
   | _ -> createloginresp; ()
  | _ -> resp; ()

val changeUserPassword: authp:prin -> unit

let changeUserPassword authp =
 let name = userid in
 let pw = password in
 let newpw = newPassword in
 let req = ChangePassword name pw newpw in
 let _ = SendMessage authp req in
  let resp = ReceiveMessage authp in
  match resp with
  | StatusMessage(status) ->
    match status with
    | Successful -> fakeprint "You have change your password"
    | Unsuccessful -> fakeprint "Something went wrong. You have
        not changed your password"
  | _ -> resp; ()

val changeUserUserId: authp:prin -> unit

let changeUserUserId authp =
 let name = userid in
 let pw = password in
 let newid = newUserId in
 let req = ChangeUserId name newid pw in
 let _ = SendMessage authp req in
 let resp = ReceiveMessage authp in
  match resp with
  | StatusMessage(status) ->
    match status with
    | Successful -> fakeprint "You have change your userid"
    | Unsuccessful -> fakeprint "Something went wrong. You have
        not changed your userid"
  | _ -> resp; ()

val identityrevoke: authp:prin -> unit

let identityrevoke authp =
 let name = userid in
 let pw = password in
 let idp = idpToRevoke in
 let req = UserRevokeIdp name pw idp in
 let _ = SendMessage authp req in
 let resp = ReceiveMessage authp in
  match resp with
  | StatusMessage(status) ->
    match status with
    | Successful -> fakeprint "You have revoked the
       identityprovider"
    | Unsuccessful -> fakeprint "Something went wrong. You have
        not revoked the identityprovider"
  | _ -> resp; ()

val addNfact: authp:prin -> unit

let addNfact authp =
 let name = userid in
 let pw = password in
 let nfact = nfactToAdd in
 let req = AddNfactor name pw nfact in
 let _ = SendMessage authp req in
 let resp = ReceiveMessage authp in
  match resp with
  | StatusMessage(status) ->
    match status with
```

```
      | Successful -> fakeprint "You have added this
        authentication method"
      | Unsuccessful -> fakeprint "Something went wrong. You have
         not added this authentication method"
    | _ -> resp; ()

val removeNfact: authp:prin -> unit

let removeNfact authp =
 let name = userid in
 let pw = password in
 let nfact = nfactToRemove in
 let req = RemoveNfactor name pw nfact in
 let _ = SendMessage authp req in
 let resp = ReceiveMessage authp in
   match resp with
   | StatusMessage(status) ->
     match status with
     | Successful -> fakeprint "You have removed this
       authentication method"
     | Unsuccessful -> fakeprint "Something went wrong. You have
        not removed this authentication method"
   | _ -> resp; ()
```

Listing 4.20: The Browser's side of the account creation and changing

This part of the *Browser* module is pretty straightforward. It specifies a lot of functions that will create the user's account and update the account by the user's wish. The nonce created when a user creates an account is used to relate the creation of an account to a user.

4.6 Introducing adversaries

In the previous section we have been focused on implementation of the protocol according to the specification. This section will introduce adversaries into the protocol verification however we have not managed in this project to incorporate dedicated adversaries. Jacob introduced adversaries in his protocol verification through an abstract program and a main function to execute a protocol run. We have adopted this way of introducing an adversary and applied it to the OpenNemID protocol as shown in listing 4.21. The difference between our way of introducing an adversary and Jacob's is ours have the Authentication Provider also. The abstract *attacker* function is a parameter to the *main* function. This means that it is able to use any function defined in the modules. However it will not be able to call any assume command, i.e. every assertion in the Service Provider, Identity Provider and Authentication Provider will succeed.

```
module Main

open SamlProtocol
open Crypto
open Serviceprovider
open Identityprovider
open Authenticationprovider

val Fork: list (unit -> unit) -> unit

let main attacker =
 Fork [ attacker;
  (fun () -> serviceprovider "serviceprovider.org" "browser" "
      identityprovider.org");
  (fun () -> identityprovider "identityprovider.org" "browser"
      "authenticationprovider.org");
  (fun () -> authenticationprovider "authenticationprovider.org
      " "identityprovider.org" "browser")]
```

Listing 4.21: Main module for introducing adversaries

It would be possible to model and mitigate known attacks on the protocol like *Man In the Middle*, *authentication replay* and *session hijacking* by modeling a browser as part of the protocol model. We have modeled the browser but we have not modeled the aforementioned attacks due to time constraints.

4.7 State of the implementation

We have implemented the Identity, Service and Authentication Provider in the OpenNemID protocol and defined their abstract implementation but they are all missing session handling. Furthermore the implementations of the cryptographic elements and networking are abstract signatures only at the moment. If time had allowed it we could have incorporated the crypto and networking experiments done by Jacob into the model which has been attached in appendix B. As the previous section explained we have not modeled dedicated adversaries as a part of the present implementation either.

Chapter 5

Evaluation

This chapter is an evaluation of the entire project. It will describe what has been accomplished during the project and what the work in this project can contribute with. It will also outline the most sensible areas to research further and discuss what related work have been done.

5.1 Project evaluation

In this report we have outlined the work of this project. The primary work in this project has been the specification of the OpenNemID protocol including a service provider, identity provider and authentication provider written in F*.

Working with a research language like F* can be a difficult challenge. However working with F* has not been quite as difficult as expected. This is because of the fact that we have extended the work done by Jacob and the F*-project is well documented with a tutorial. This means that we could always use Jacob's work as a reference and the compiler was easily to set up because of his work. We both have a little experience with ML and the functional oriented language F# which F* is based on - these facts also helped us during the development of this project. Furthermore we both also have a strong background in C# and .NET.

The model of OpenNemID we have presented can be evaluated against common software development and software architecture principles. What should be noted as interesting is not the design of modules but the kind of style a language like F* imposes on the developer. F* enforces the principle of programming to an interface. F* takes the principle further to *design by contract* known from other languages. Functional language with immutability as default makes it easier for developers to design components that holds no state. This makes the scalability of system across multiple processors or computers a lot easier.

5.2 Related work

We have extended the work done by Jacob Højgaard [1] in his masters thesis *Securing Single Sign-On System With Executable Models.* His work has had a huge impact on this project as we have stated throughout the report. The impact has been both in understanding the language F* and designing and specifying the OpenNemID protocol. In the masters thesis *Using static analysis to validate SAML Protocols* written by Hansen and Skriver [3] they analyze and formalize the SAML login protocol. Their analysis goes towards the older version 1.1 SAML protocol. In their analysis they recommend mandating of HTTPS network transport. This has been incorporated in the OIOSAML specification done by Jacob.

5.3 Threats to validity

When working in the area of security protocols there will be a lot of threats to validity. Therefore it is important to scrutinize the work of this report.

As stated earlier we both have a little experience with the functional programming language F# which F* is based on. However since we are no experts in F# and have worked mostly with object oriented languages there is a good chance that the capabilities of F* has not been fully utilized. Furthermore we have not worked with any formal specification tools before beginning this project.

It is also important to remember that the F* compiler is still a α-release. Microsoft Research has used it to verify 20.000 lines of code [7] but the results should still be treated with some degree of caution. Still the concepts in F* are based on are derived from the work in the previous projects, F7 and Fine. In these projects the concepts have been investigated thoroughly.

Lastly it is worth noting we have only touched the subject of modeling adversaries very briefly. This leaves room for uncertainty and therefore would be a natural course to take for further research.

5.4 Further research

As we have mentioned the modeling of adversaries is the natural path to take because this is required to make a more robust framework for protocol verification. In section 4.6 we described that we have modeled a browser but we have not modeled adversaries as part of the browser. This could be a way to continue the research. We mentioned briefly that F* could be compiled to JavaScript. The compilation of F* to JavaScript is interesting because NemID uses a Java applet which forces all the user to install a Java browser plug-in. Given the facts that there have been many security issues reported lately with the Java browser plug-in and that DanID are working on a version of NemID in JavaScript F*'s compilation to JavaScript is very to have in mind. This means that a specification in F* could be compiled to JavaScript and used on the web and thereby eliminate the large security risk involved with using the Java applet. Further research could also be into the server side of the Authentication Provider (as we have called it). As we have specified the Authentication Provider the next step would be to actually creating the server side of it so it could be an executable protocol on the web.

Our design of OpenNemID included security profiles. Due to time constraints this has not been specified in the current specification of OpenNemID in F*. Continuing the research could be done by extending the protocol specification to support the functionality of security profiles.

5.5 Conclusion

In this report we have shown how the use of programming language F* for implementing security protocols can help proving the security properties of the protocol.

In this project we have extended the work of Jacob Højgaard whom has used the Danish governmental Single Sign-On federation - Nemlog-in as an example. Nemlog-in is a protocol based on the SAML standard for exchanging security tokens. In the Nemlog-in protocol the authentication part has been done by NemID. This project has extended and modified the protocol specification done by Jacob to specify the authentication part as well. A protocol we refer to as OpenNemID. OpenNemID has been described and outlined including the registration of users and logging in of users.

In this report we have formally specified key parts of the OpenNemID protocol using the F* programming language to the extent possible and used it to secure key properties such as signing of the messages exchanged over the network.

Since there has only been formally specified key security properties we have described a way forward for securing other properties. We have suggested the use of F* to model known attacks as part of the protocol test suite.

In conclusion a programming language such as F* can be used to create an executable model of a security protocol. This ensures the security features mandated by the protocol.

Bibliography

[1] Jakob Højgaard: *Securing Single Sign-On System With Executable Models.* Masters thesis, IT University of Copenhagen, 2013.

[2] David Basin, Patrick Schaller, Michael Schläpfer: *Applied Information Security - A Hands-on Approach.* Springer, Berlin Heidelberg, 2011.

[3] Steffen M. Hansen and Jakob Skriver: *Using Static Analysis to Validate SAML Protocols.*
Masters thesis, Technical University of Denmark, 2004.

[4] The WAYF secretariat: *Trusted third party based ID federation, lowering the bar for connecting and enhancing privacy.* Explanation of WAYF, 2009.

[5] Danny Dolev and Andrew C. Yao: *On the security of public key protocols.* Technical report, Stanford, CA, USA 1981.

[6] Auguste Kerckhoffs: *La cryptographie militaire.* Journal des Sciences Militaires, IX:538, 161191, 1883

[7] Microsoft Research: *F*: A Verifying ML Compiler for Distributed Programming.* `http://research.microsoft.com/en-us/projects/fstar/`

[8] Microsoft Research: *F* guide* `http://rise4fun.com/FStar/tutorial/guide`

[9] ProVerif: *Cryptographic protocol verifier in the formal model.* `http://prosecco.gforge.inria.fr/personal/bblanche/proverif/`

[10] W3.org: *Web Cryptography API.* `http://www.w3.org/TR/WebCryptoAPI/`

[11] JSCert: *Certified JavaScript.* `http://jscert.org/index.html`

[12] XOR-ProVerif: *Tool for analyzing protocols with XOR.* `http://www.infsec.uni-trier.de/publications/paper/KuestersTruderung-CCS-2008.pdf`

[13] DH-ProVerif: *Tool for analyzing protocol with Diffie-Hellman exponentiation.* `http://www.infsec.uni-trier.de/publications/paper/KuestersTruderung-CSF-2009.pdf`

[14] CSProto: *Tool for analyzing contract signing protocols* http://www.infsec.uni-trier.de/publications/paper/KaehlerKuestersWilke-TOCL-2010.pdf

[15] Sergio Maffeis: *An expert in computer security.* Various publications on web security, security protocols and so on. http://www.doc.ic.ac.uk/~maffeis/

[16] NaCl: *Networking and Cryptography library.* http://nacl.cr.yp.to

[17] Sodium: *A new cross-compilable Cryptography library.* A variation of the NaCl library. http://labs.umbrella.com/2013/03/06/announcing-sodium-a-new-cryptographic-library/

[18] Digitaliseringsstyrrelsen: *Den digitale vej til fremtiden velfærd - Den fællesoffentlige digitaliseringsstrategi 2011-2015.* http://www.digst.dk/Home/Digitaliseringsstrategi/~/media/Files/Digitaliseringsstrategi/Digitale_vej_til_fremtidens_velf%C3%A6rd.ashx

[19] Nets DanID: *Forretningsbetingelser MOCES DanID april 2012.* https://www.nets-danid.dk/produkter/nemid_medarbejdersignatur/vilkaar/forretningsbetingelser_nemid_administrator/Forretningsbetingelser_MOCES_DanID_april_2012.pdf

[20] Nets-Danid: *NemID guide* https://www.nets-danid.dk/produkter/for_tjenesteudbydere/nemid_tjenesteudbyder/nemid_tjenesteudbyder_support/tjenesteudbyderpakken/

[21] Dr. Joseph Roland Kiniry: *Parliamentary Statement for Bill L-132* http://www.ft.dk/samling/20121/almdel/kou/bilag/70/1225808.pdf

[22] NemID: *Om NemID - Sikkerhed* https://www.nemid.nu/dk-da/om_nemid/sikkerhed/

[23] Kaspersky: *Oracle Java surpasses Adobe Reader as the most frequently exploited software* http://www.kaspersky.com/about/news/virus/2012/Oracle_Java_surpasses_Adobe_Reader_as_the_most_frequently_exploited_software

[24] RFC 6238: *TOTP: Time-Based One-Time Password Algorithm* http://tools.ietf.org/html/rfc6238

[25] OpenID: *OpenID Foundation website* http://openid.net/

[26] Yubico: *Two-factor authentication with the YubiKey* http://www.yubico.com/

Appendix A

SAML Assertion and Messages

Appendix B

Source code examples by Jacob Højgaard