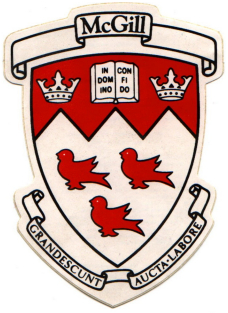

Advanced Applications for e-ID Cards in Flanders

ADAPID Deliverable D12

Framework II

Bart De Decker, Jorn Lapon, Mohamed Layouni, Raphael Mannadiar,
Vincent Naessens, Hans Vangheluwe, Pieter Verhaeghe,
Kristof Verslype (Ed.).

December 2009

Executive Summary

This deliverable is part of a work package within the Adapid project. The work package aims to deliver a software framework that allows application developers to easily create privacy preserving applications. This deliverable is the second step in the development of this framework.

In summary, the developed framework handlers aim at providing the application developer a uniform interface to set up connections that can be privacy enhancing, to issue, receive and use credentials that might be privacy preserving and it offers a way to store the related data.

The deliverable sketches an extensive and extensible framework consisting of lower level handlers and higher level managers. The focus of this deliverable was mainly on the handler level, while the manager level is mainly seen as future work. The handlers need to be implemented and a providers, which can be plugged into the framewok, has been made. This providers contains multiple implmentations for the handlers interfaces defined in the framework itself.

A validation based on an ePoll and an eTicketing system has been made, which confirms the usefulness of the framework. However, there are still a lot of open issue that need to be examined in the future.

One chapter is devoted to modelling and synthesizing privacy-presering applications. The aim of this apporach is to develop privacy preserving applications by modelling them in a domain specic model. Based on this model, code could be generated, for instance, code that uses the framework interfaces.

Contents

List of Figures

Listings

Chapter 1

Introduction

The digitalilization of our society brought a lot of benefits. However, privacy of the user is more and more at stake. The awareness in the society w.r.t. privacy is rising mainly due to a number of press articles in which for instance millions of personal records were simply lost. Some political parties even have the right of privacy as part of their programm as a reaction on the evolutions in recent years.

Nowadays, companies can compose giant databases about their customers and can even link these databases together. This is for instance the case after a merger of companies. Two typical examples of companies who really know a lot about you are Google and Facebook. Imagine that all your personal data is stolen and sold or made public? Especially towards health related data, privacy is an important issue, since these data are extremely sensitive. Image for instance that you want to get an life insurance, but all the insurance companies know some crucial details about health status. Or imagine finding a job if you facebook live is available to your future employer.

Also, the use of eID cards is nowadays in most cases very privacy unfriendly. If a company asks you to inserts your Belgian eID card into a card reader connected to his computer, he can simply read out your name, address, picture, data of birth, etc. even without having to insert your pin code.

The user should be given the control of deciding what personal information he is willing to disclose to whom. Technologies are (being) developed that 1) provide the user anonymity at network level and 2) that allow the user to disclose in a fine grained way the properties (s)he is willing to disclose. For instance, instead of disclosing the exact date of birth in order to prove that one is older than 18, the user could prove only that he is older than 18. Secondly, instead of disclosing data under an identifier such as his SSN, he could disclose properties under a pseudonym. This is also valuable to the companies, since securing their databases against breaches and insider attacks has shown to be really though and if sensitive personal data about their customers is lost, it can cost the company a lot of money.

The privacy enhancing technologies are not easy to use and each technology has its own interface and peculiarities. Hence, it will cost the application developer a lot of time to make his applications privacy friendly, which increases the probability that 1) the privacy issues will be omitted or 2) if they are not omitted, the privacy is still compromised due to a bad use of the privacy preserving technologies. Moreover, the user will still not be able to keep track of the personal information that has been disclosed to whom and under which pseudonym.

The aim of this framework is to provide a means towards the application developer to easily integrate privacy in client-server applications. More exactly; the framework wants to offer the application developer the means to control in a fine grained way the data that has been disclosed to whom. A uniform interface is provided to set up and listen for connections, which might or might not give the user a certain degree of anonymity. On the other hand, a uniform interface offers the application developer the possibility to use whatever credential technology that is the most appropriate. This makes it very easy to switch to another, more efficient or more privacy-preserving technology.

On the other hand, we can expect that the user will have a lof of digital credential in the future; one for his football club, one for on his eID card, his subscription to the cinema, his medical insurance certificate, his driving permission, etc. The framework wants to offer a means to manage these credentials, although the credentials can be stored or cached on different places, even remotely.

The framework offers support to keep track of what information has been disclosed to whom under what pseudonym. The user can, assisted by the framework, decide whether he is willing to disclose more information, or wheter he should change to another pseudonym or instead abort the action. The framework aims at informing the user about his current anonymity towards other parties and about the effect of disclosing certain information.

The framework provides "a uniform interface to facilitate the development of privacy preserving applications". It enables a programmer with extensive to basic understanding of cryptographic- and security-related concepts to implement privacy preserving applications. The framework can thus – simplistically – be viewed as a means to enable programmers to do more with less hassle. From here arises an interesting question – especially under the premise that cryptography and/or security experts are harder to come by than able programmers – : what can be done to enable domain experts to do more with less hassle? More specifically, what can be done to enable cryptography and/or security experts with little to no programming capabilities to develop privacy preserving applications?

Domain-Specific Modelling (DSM) is a relatively young discipline whereby applications[1] are modelled at a high-level of abstraction using constructs that are tightly coupled to some restricted domain's concepts [25, 19, 6]. Subsequent *Model Transformations* transform the models into a collection of low-level artifacts that form the final applications. The advantages of using model-driven approaches as opposed to code-centric approaches to application development are numerous; some of them are listed below.

Although a lot of work has been devoted to the framework, there are still a lot of issues that can be covered or extended. This deliverable gives an overview of the work that has been done so far.

This deliverables continues as follows; In the next chapter, the preliminaries are given; the framework principles, important cryptographic techniques and credential techniques. In chapter 3, the actual framework is explained and 4 gives a validation bases on applications

[1] A wide variety of artifacts can be modelled ranging from configuration files, to data schemas, to protocols, to partial or complete applications. For brevity, we restrict our discussion of DSM to the modelling and synthesis of applications.

discussed in the appendix. Chapter 5 discusses domain specic modelling. Finally,in chapter 6, the conclusions and future work are given.

Chapter 2

Preliminaries

2.1 Framework General Definition and Principles

In this section, we explain what a framework exactly is, and what principles must be fullfilled.

2.1.1 Definition

"A software framework is a reusable design for a software system (or subsystem). This is expressed as a set of abstract classes and the way their instances collaborate for a specific type of software." - Johnson and Foote 1988; Deutsch 1989 [24]

According to Pree [29], software frameworks consists of frozen spots and hot spots. On the one hand, frozen spots define the overall architecture of a software system: its basic components and the relationships between them. These remain unchanged (frozen) in any instantiation of the application framework. On the other hand, hot spots represent those parts where the programmers using the framework add their own code to enhance/extend the functionality specific to their own project.

A framework should allow the developers to spend more time concentrating on the business-specific problem rather than on the code. Also a Framework will limit the choices during development, so it increases productivity, specifically in big and complex systems. A framework should eliminate the effort of continuous re-discovery, re-invention of concepts and their re-development.

A software framework can be geared toward building graphical editors for different domains like artistic drawing, music composition, and mechanical CAD. Another software framework can help build compilers for different programming languages and target machines. Yet another might help build financial modeling applications or decision support systems. There are frameworks for multimedia, web applications, and even communicating between different systems.

2.1.2 General Framework Requirements

In this subsection, general guidelines for a good framework are presented.

The primary benefits of Object Oriented application frameworks stem from the modularity, reusability, extensibility, and inversion of control they provide to developers, as described in [20] and repeated below:

- **Modularity.** Frameworks enhance modularity by encapsulating volatile implementation details behind stable interfaces. Framework modularity helps improve software quality by localizing the impact of design and implementation changes. This localization reduces the effort required to understand and maintain existing software.
- **Reusability.** The stable interfaces provided by frameworks enhance reusability by defining generic components that can be reapplied to create new applications. Framework reusability leverages the domain knowledge and prior effort of experienced developers in order to avoid re-creating and re-validating common solutions to recurring application requirements and software design challenges. Reuse of framework components can yield substantial improvements in programmer productivity, as well as enhance the quality, performance, reliability and interoperability of software.
- **Extensibility.** A framework enhances extensibility by providing explicit hook methods [Pree:94] that allow applications to extend its stable interfaces. Hook methods systematically decouple the stable interfaces and behaviors of an application domain from the variations required by instantiations of an application in a particular context. Framework extensibility is essential to ensure timely customization of new application services and features.
- **Inversion of control.** The run-time architecture of a framework is characterized by an "inversion of control." This architecture enables canonical application processing steps to be customized by event handler objects that are invoked via the framework's reactive dispatching mechanism. When events occur, the framework's dispatcher reacts by invoking hook methods on pre-registered handler objects, which perform application-specific processing on the events. Inversion of control allows the framework (rather than each application) to determine which set of application-specific methods to invoke in response to external events (such as window messages arriving from end-users or packets arriving on communication ports).

Another requirement is **usability**, which is indeed a broad term which involves multiple elements. This means that the framework should be *sufficiently easy to learn* by application developers: learning to use the the framework must not be harder than learning to use the individual technologies that are required to develop the applications. Learning how to work with the framework must not require a huge amount of effort, otherwise it will not be used at all. Therefore, the API must be made as intuitive as possible. To quote Einstein: "Things should be made as simple as possible, but not any simpler". The framework should be *easy to configure, deploy and maintain* once deployed. It is unavoidable that the framework will need to be updated. This must be possible with minimal changes by the application developers. Usability also implies that the framework does *not* involve a big computational or storage *burden*. Usability also means that a good, detailed and clear *documentation* should be available. Although this is often seen as lower-priority task, it is indispensable in gaining popularity amongst application developers.

Testing and validation is an indispensable part of framework development. Due to the high level of abstraction, testing the framework apart will be a though job. It can as well be

tested using concrete use case implementations, however if a bug is found, it can be located in either the framework or the application built upon the framework. However, *validation* of the framework using concrete use cases is necessary to test the completeness of the functionality of the framework, to test its user-friendliness, etc. Based on the feedback, a next iteration is possible.

Indeed, the framework's functionality are abstractions derived from concrete use cases. Based on those concrete examples, common functionality is grouped and the corresponding methods are refactored to generic methods. This generalization is an iterative process and is based on the feedback given by the experts of the different use cases. Analyzing and implementing those example use cases is a large fraction of the cost of the project.

2.1.3 Providers

The framework as such cannot be used by the application developer. Therefore, an implementation of one or more components is necessary. Three properties must be satisfied: technology independence, implementation independence and multiple instances:

- **Technology independence.** It must be possible to add or remove technologies to the framework in a transparent and easy way for the application developer. E.g. if the framework only offers support for the Belgian eID card, it must be possible to change the used technology to the Swedish eID card.
- **Implementation independence.** The possibility must be offered to easily change the implementation of a technology supported by the framework. E.g. if implementation A is plugged in offering the possibility to show the Belgian eID to others, it must be possible to change to another implementation B doing the same. This can be useful e.g. if security breaches are detected, or if more efficient implementations are offered. This allows lightweight software implementations on mobile phones, more heavy weight software implementations on desktop computers and fast hardware implementation on servers.
- **Multiple instances.** In one framework, it must be possible to use multiple technologies for the same interface at the same time. This way, the support for the Belgian and Swedish eID card is possible at the same time. One must be able to choose which technology to use (e.g. to select the credential type to issue). If multiple implementations can be plugged into the framework, one must be able to choose the implementation (the one that is considered as the most thrustworthy one, the most efficient one, etc). One must be able to add or delete and choose technologies and implementations in an easy way, preferably at runtime.

Multiple instances allow the Adapid framework to support classical X.509, Idemix and U-Prove credentials and multiple eId cards at the same time. Hundred percent technology independence is impossible in our framework as different technologies have different properties. At least the different technologies must be usable by the application developer in a similar way.

To obtain these properties, so called providers can be used. A provider contains one or more implementations of one or more technologies that fit into the framework. Thus, at the one hand, we have the framework, offering the API, and at the other hand, we have the

providers, offering the implementations. Different providers must be plugable at the same time into the framework without conflicting.

2.1.4 Sensitive Data Representation

In a security framework, it makes sense to have two representations of sensitive data (e.g. secret keys) outside the framework. *Opaque representations* allow one to access only the non sensitive data (e.g. name), but disables the possibility to use the sensitive data outside the framework. The full functionality can only be used inside the framework. This is the more secure and default representation within the framework context. *Transparent representations* on the other hand do not have these restrictions and allow to export the sensitive data. This representation will typically be a global standard representation and thus allows interoperability.

In the ADAPID framework, the sensitive data consists of two types of sensitive data: secret keys and personal data. The former is needed to use credentials while the latter is contained in credentials or in evidence of actions (transcripts). By default, an opaque representation will be used to enhance security, while the possibility of transparent representations must evidently be offered to enhance interoperability with other systems able to handle the data (e.g. X.509 certificates).

2.1.5 Patterns

In software engineering (or computer science), a design pattern [23] is a general repeatable solution to a commonly occurring problem in software design. A design pattern is not a finished design that can be transformed directly into code. It is a description or template for how to solve a problem that can be used in many different situations. Object-oriented design patterns typically show relationships and interactions between classes or objects, without specifying the final application classes or objects that are involved. Algorithms are not thought of as design patterns, since they solve computational problems rather than design problems. In designing frameworks, the use of patterns is of utmost importance in obtaining maximal flexibility and extensibility. In this section, a few examples of how pattern can be used in the design of frameworks are given.

A potentially important design pattern is the *Bridge Pattern*. It decouples an abstraction from its implementation, so that the two can vary independently. This is important if a framework has to offer uniform APIs, while the underlying algoritms and implementation can vary.

The *Observer pattern* offers a mechanism to keep related components consistent, without having a close coupling between these components, as this increases complexity and decreases reusability.

As a final example, setting up a connection to a server can be rather tedious as complex interfaces and network operations are involved. The *Service Locator pattern* hides this complexity

Other patterns exist as well. We refer to [23] for an overview of useful patterns.

2.2 Cryptographic Building Blocks

Throughout the deliverable, often, the following notation will be used: $X \leftrightarrows Y : (r_x; r_y) \leftarrow f(a; b; c)$. a and b are inputs known to and given by X and Y respectively. c is common input. r_x is the result at the side of X, r_y at the side of Y.

2.2.1 Commitments

A **commitment** can be seen as the digital analogue of a "non-transparent sealed envelope". It enables a committer to hide a set of attributes (non-transparency property), while at the same time preventing him from changing these values after commitment (sealed property). The committing party can prove properties of the attributes embedded in the commitment.

- *(Com, OpenInfo) ← commit(Attribute(s))*
  A new commitment *Com* is generated as well as secret information *OpenInfo* containing, among others, the attributes embedded in *Com*. This information can be used to prove properties about the attributes.
- $\mathcal{P} \leftarrow \mathcal{V}$*: prove(OpenInfo; ∅; Com, pred(attrs))*
  The public input to this protocol is both a commitment *Com* and a boolean predicate *pred* concerning *com*'s attributes. If $\mathcal{V}$ accepts the proof, $\mathcal{V}$ is convinced that $\mathcal{P}$ knows *OpenInfo* belonging to *Com*, and that *Com*'s attributes satisfy predicate *pred*. Note that *prove* is usually an interactive protocol, but it can be made non-interactive.

A simple example of an application of commitments is secure coin-flipping. Suppose, Bob and Alice want to resolve some dilemma by coin flipping. If they are physically in the same place, the procedure would be (1) Bob "calls" the coin flip, (2) Alice flips the coin and (3) if Bob's call is correct, he wins, otherwise Alice wins. If they are not in the same place, however, this does not work, since Bob has to trust Alice to report the outcome of the coin flip correctly. With commitments, a similar procedure can be constructed: (1) Bob "calls" the coin flip but only tells Alice a commitment to the call, (2) Alice flips the coin and reports the result to Bob, (3) Bob reveals what he committed to, (4) if Bob's call matches the result, Bob wins, otherwise, Alice wins.

2.2.2 Proof of knowledge and Zero-knowledge proof

A *proof of knowledge* is a proof in which the prover succeeds in 'convincing' a verifier that he knows something. Assume a set X of publicly known elements; for each $x \in X$, there exists a witness $w \in W$ for which the relation $R(x, w)$ is true. The prover $\mathcal{P}$ can then prove to $\mathcal{V}$ that he knows w without revealing it to $\mathcal{V}$.

$$\mathcal{P} \leftarrow \mathcal{V} : PK\{(w) : R(x, w)\} \tag{2.1}$$

where x is the public input to the protocol, and w the corresponding secret witness w, the knowledge of which is proven by $\mathcal{P}$ to $\mathcal{V}$.

A *zero-knowledge proof* is an interactive protocol in which a party $\mathcal{P}$ proves to another party $\mathcal{V}$ that a statement is true, without revealing anything other than the veracity of the statement.

$$\mathcal{P} \leftarrow \mathcal{V} : ZKP\{(x) : pred(x)\} \tag{2.2}$$

where x is a secret value of which predicate *pred* is proven to $\mathcal{V}$. $\mathcal{V}$ does not gain any other information about x except that $pred(x)$ is true.

Example of Proof of knowledge and Zero-Knowledge proof An authentication protocol based on public key cryptography usually uses a proof-of-knowledge protocol, in which the prover proves that he knows the private key:

$\mathcal{P}$		$\mathcal{V}$
		choose random challenge *rnd* $c \leftarrow$ encrypt(PubKey, *rnd*)
	$\xleftarrow{c}$	
$r \leftarrow$ decrypt(PrivKey, c)		
	$\xrightarrow{r}$	
		if ($r == rnd$) then SUCCESSFUL else FAILED

$\mathcal{P}$ proves that he knows the private key (PrivKey) that corresponds to the public key (PubKey). This is abbreviated as follows:

$$\mathcal{P} \rightarrow \mathcal{V}\text{: PK}\{\text{(PrivKey): } rnd = \text{decrypt(PrivKey, encrypt(PubKey, } rnd\text{))}\}$$

Although the private key (PrivKey) is not revealed to $\mathcal{V}$, a dishonest $\mathcal{V}$ can learn something about the private key by sending random data to $\mathcal{P}$ and have $\mathcal{P}$ decrypt that data ($\mathcal{P}$ is used as an oracle by $\mathcal{V}$ for decrypting data).

The previous protocol can easily be transformed into a zero-knowledge proof:

$\mathcal{P}$		$\mathcal{V}$
		choose random challenge *rnd* $c \leftarrow$ encrypt(PubKey, *rnd*)
	$\xleftarrow{c}$	
$r \leftarrow$ decrypt(PrivKey, c) *(com, openinfo)* $\leftarrow$ commit(r)		
	$\xrightarrow{com}$	
	$\xleftarrow{rnd}$	
if ($rnd \neq r$) then ABORT		
	$\xrightarrow{openinfo}$	
		$r \leftarrow$ openCommit(*com, openinfo*) if ($r == rnd$) then SUCCESSFUL else FAILED

Here $\mathcal{P}$ first sends a commitment to the challenge, and when $\mathcal{V}$ reveals the challenge, $\mathcal{P}$ opens the commitment for $\mathcal{V}$. A dishonest $\mathcal{V}$ is not able to send the challenge in cleartext (in the third message) if the first message was random data.

$\mathcal{P}$ proves (in zero-knowledge) that he knows the private key (PrivKey) that corresponds to the public key (PubKey). This is abbreviated as follows:

$$\mathcal{P} \rightarrow \mathcal{V}\text{: ZKP}\{\text{(PrivKey): } rnd = \text{decrypt(PrivKey, encrypt(PubKey, } rnd\text{))}\}$$

Other examples of zero-knowledge proves include proving that an attribute embedded in a commitment or credential (cfr. section 2.3), is greater than a certain value (e.g. the age-attribute is greater than 18) or that the attribute lies in a fixed interval (e.g. the age-attribute lies in [18,35]).

2.2.3 Verifiable encryptions

Verifiable encryptions have all the characteristics of regular (public key) encryptions. Additionally, they enable the creator $\mathcal{P}$ to demonstrate properties of the encrypted plain text. As an example, $\mathcal{P}$ can prove to $\mathcal{V}$ that the encrypted plain text is encoded as an attribute in a commitment or credential.

$$c = VE(PubKey, x) \tag{2.3}$$

c refers to the cipher text, x refers to the plain text, $PubKey$ is the public key of a TTP. Note that the corresponding private key may not be known to $\mathcal{V}$ nor $\mathcal{P}$.

Verifiable encryptions can be used to implement accountability in anonymous transactions. The customer will send a verifiable encryption of his pseudonym (or identity) to the service provider, thereby proving that the encrypted plain text is correctly formed (e.g. is equal to the pseudonym embedded in a credential). The service provider is assured that when abuse is detected, the TTP will be able to reveal the pseudonym (or identity) of the anonymous customer.

2.3 Credential systems

2.3.1 Belgian eID Card

This section gives an overview of the current Belgian eID technology. A more elaborate description can be found in [17, 18].

Contents of the Belgian eID card

Private information such as the owner's name, birthdate and -place, address, digital picture and National Registration Number is stored in three separate files: an identity file, an address file and a picture file. The files are signed by the National Registration Bureau (NRB). The National Registration Number (NRN) is a unique nation-wide identification number that is assigned to each natural person.

Two key pairs are stored on the eID card. One key pair is used for authentication, the other is used is for signing. The (qualified) e-signatures are legally binding. The public keys are embedded in a certificate which also contains the NRN and the name of the card holder. The private keys are stored in a tamper-proof part of the chip and can only be activated (not

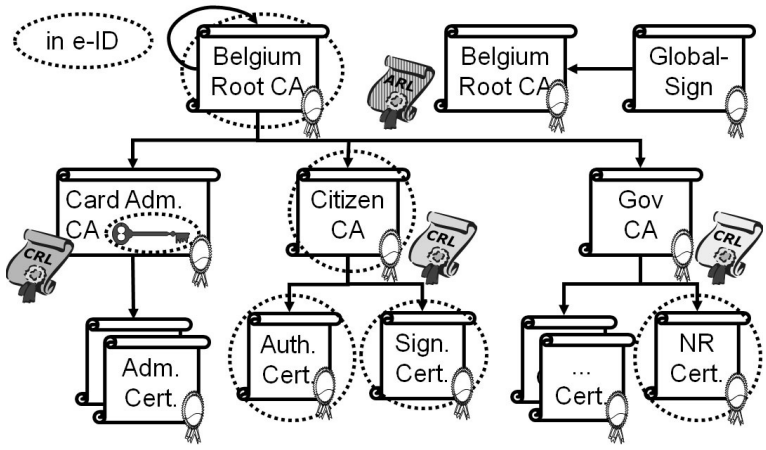

Figure 2.1: Belgian Public Key Infrastructure

read) with a PIN code. Authentication is single sign-on, i.e. the PIN code is only required for the first authentication. For signing, a PIN code is needed for each signature[33].

Belgian Public Key Infrastructure

The certificates on the eID card are part of a larger hierarchical infrastructure, the Belgian Public Key Infrastructure (be-PKI) [30]. The hierarchy is illustrated in figure 2.1. The citizen's signature and authentication certificates are issued by a *Citizen_CA* which is certified by the *Belgium_Root_CA*. Other governmental CAs such as the *Card_Admin_CA* and *Government_CA* also have certificates issued by the *Belgium_Root_CA*. The former can update the eID card. The latter certifies the National Registration Bureau (NRB) which signs the identity and address file and offers other services in the public sector. The *Belgium_Root_CA* has two certificates. The first is a self-signed certificate, that allows for offline validation of the signature and authentication certificates on the eID card. The second certificate is issued by GlobalSign. The latter is typically known to popular applications (such as browsers) and allows for the automatic validation of electronic signatures. The PKI provides Authority Revocation Lists (ARL) and Certificate Revocation Lists (CRL) [1] that keeps the serial numbers of the revoked certificates.

Official middleware

The cryptographic functionalities in the Belgian eID card are accessed through middleware [3]. Applications typically interact with the card via a simple API [31] offered by this middleware. If a document needs to be signed, the middleware passes a hash of the document to the card. Similarly, a hash of the challenge is passed to the card for authentication purposes. When an application wants to authenticate or sign a document with the eID card, the middleware requests the user to enter his PIN code. The middleware can also verify the validity of the certificates (using CRL or OCSP). It is important to note that the use of the official middleware is not mandatory. Several alternatives, developed by different companies, are available.

2.3.2 Classical X.509 certificates

Classical X.509 certificates are well known and are basically a set of attributes signed by a trusted party. Three protocols are relevant in this deliverable.

- $E \leftrightarrows I : (cert_E, SK_E; cert_E) \leftarrow \mathsf{issueCert}(cert_I; \varnothing; atts, SK_I)$. I issues a certificate $cert_E$ with secret key SK_E to entity E by signing a set of attributes $atts$ with his private key SK_I. We assume that the key pair (PK_E, SK_E) is generated as part of this protocol.
- $P \leftrightarrows V : \mathsf{authenticate}(cert_P; SK_P; \varnothing)$. P authenticates towards V using his private key SK_P.
- $U : sig \leftarrow \mathsf{sig}(SK_U, msg)$. Sign a message msg using secret key SK_U .

2.3.3 Pseudonym Certificates

Pseudonym certificates [4] are traditional certificates where the identity information is replaced by a pseudonym. The certificate states that the identity of the user referred to by that pseudonym and the properties certified in the certificate have been verified by the issuer. Different shows of the same certificate are linkable, which can undermine anonymity. The same functions as for X.509 certificates can also be applied for pseudonym certificates.

Enhanced Pseudonymous Certificates.

We further extend the privacy without requiring considerable computational capabilities by replacing each certificate attribute *att* that contains personal properties (date of birth, social security number, etc.) by $H(att, \text{RAND})$. Showing such an enhanced pseudonym certificate thus only reveals personal data if the owner of the certificate also discloses the corresponding (*att*, RAND) tuple to the verifier. Evidently, the linkability issue persists.

2.3.4 Anonymous credentials

Anonymous credential systems [14, 13, 11, 7] allow for anonymous yet accountable transactions between users and organisations. They are capability based, user-centric mechanisms in which users can prove statements about themselves and their relations with others *anonymously*. Moreover, selective disclosure allows the user to reveal only a limited set of properties of the attributes embedded in the credential: e.g. a credential with the user's date of birth as an attribute can be used to prove that the owner is over 18 without disclosing the exact date of birth or other attributes.

Two anonymous credential systems are being implemented.

- *U-Prove* credentials are pseudo-anonymous, even to the issuer; selective disclosure is possible, but different shows of the same credential are linkable. However, a credential can easily be reissued.
- *Idemix* is more flexible and can be used with or without pseudonyms. Multiple credential shows are unlinkable if no uniquely identifying attribute data are revealed. Therefore, Idemix is used in the rest of this paper.

The relevant simplified protocols that apply to anonymous credentials are:

- $U \leftrightarrows I$: Cred $\leftarrow$ issueCred$(coms; atts_U, opens, cert_I; atts_I, SK_I)$. I issues to U a credential. The credential attribute values are either chosen by U and hidden for I ($atts_U$), or chosen by I ($atts_I$). Committed values can be included in the credential, allowing U to hide the actual values for I, while still being able to prove properties about those values. The corresponding commitments and opening infos are *coms* and *opens*.
- $U \leftrightarrows V$: $proof \leftarrow$ showCred$(coms, \text{properties}; \text{Cred}, opens; \varnothing)\{\text{Msg}\}$. U proves to V the possession of a valid credential Cred. U can selectively disclose credential attributes or properties thereof (described in *properties*). These properties can involve a set of committed values *coms* with opening info set *opens*. U may decide to sign a message Msg with his credential, creating a provable link between the proof and the message.

Additionally, proofs resulting from a showCred() protocol can be deanonymizable by a predetermined trusted third party. Anonymous credentials can be issued to be shown either once, a predetermined limited number of times or an unlimited number of times in total, independent of the services that are contacted.

Simillarly to commitments, relationships with values in verifiable encryptions can be proven.

Chapter 3

Framework

3.1 Overview

In general, the aim of the framework is to offer a *uniform interface* to facilitate the development of privacy preserving applications. This way, the complexity of the underlying building blocks is hidden for the application developer. Hence, developing a privacy preserving application becomes faster and migrating to new (more privacy enhancing) technologies becomes easier. The focus of the framework is on certifying personal data and on managing and disclosing certified personal data. Furthermore, the client-server model is considered.

The general architecture of the framework is shown in figure 3.1. The middle layer is the actual framework. It consists of several handlers and managers. The handlers are abstract classes, providing a uniform interface to a family of similar technologies; technologies that are implementations of the same concept. Some of these technologies might have privacy enhancing properties, while others don't. For instance, not all credentials are privacy enhancing.

The lowest layer, i.e. the provider layer, contains one or more providers. Each provider contains implementations for at least one handler interface, resulting in a concrete handler object. Hence, for each implementation in the provider layer, we have a separate handler. Multiple providers can offer implementations for the same technology and a provider can contain multiple implementations for the same abstract handler interface.

On top of the handlers, the managers are found. A manager is a singleton; for each manager type, there is only one instance. Managers provide higher level functionality compared to the handlers and, therefore, they use the underlying handlers. On the contrary to handlers, managers are fully implemented in the framework itself and do not need to be implemented by a provider.

On top of the framework layer, the framework entry point `Framework` is found. Here, applications can get references to the different managers. The managers in turn keep track of the different available handlers. Secondly, it loads and applies the general configuration settings. Thirdly, it can enforce user authentication towards the framework; this can be done using a simple password, but also other authentication mechanisms such as eID authentication are possible.

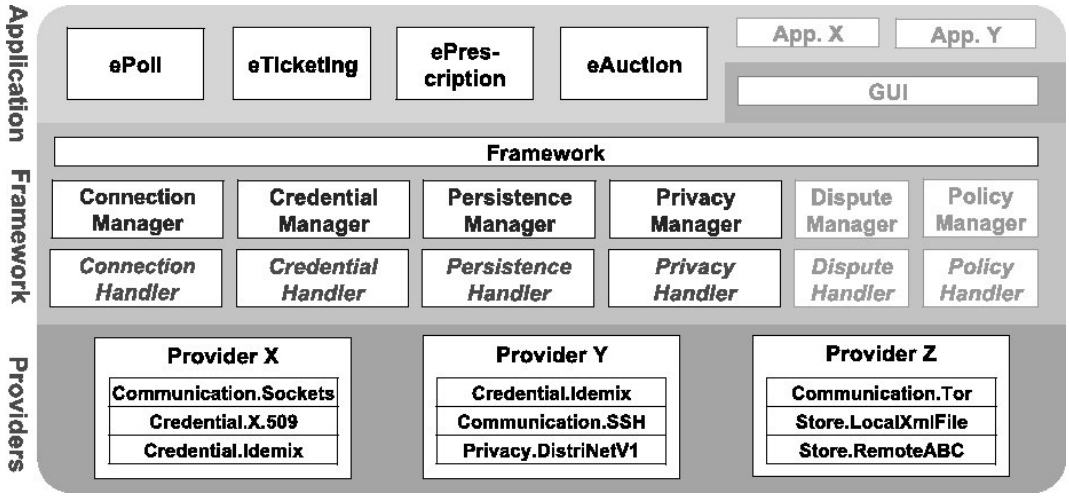

Figure 3.1: High level architecture of the framework.

3.1.1 Design

An overview of the general design can be found in figure 3.2.

- `Framework`. The central class loads at startup the different handler implementations that are provided by the available providers. As part of the constructor of the `Framework` class, the user might be required to authenticate.
- `Handler`. Each `Handler` component in the framework defines a generic interface to a category of technologies (credentials, conections, etc.). An implementation is necessary before a component can be used. This implementation can be provided by `Provider` packages. After one or more providers are plugged into the framework, the managers can automatically select the proper implementation when necessary.
- `Provider`. This interface needs to be implemented for each provider; meaning that for each handler implementation provided by a provider, an `Implementation` object is added to the concrete class in that concrete provider implementing the `Provider` interface. The information in these `Implementation` objects is used in the framework to forward API calls from applications to the corresponding classes (with the actual implementation) in the provider package.
- `Manager`. In the current framework, four main managers are present; the credential, persistence, connection and privacy manager. Each of these managers is a singleton which keeps track of the available corresponding handler implementations.

The framework provides some other classes (see figure 3.3) from which framework specific classes such as `Credential` and `ConnectionParameters` (see later), will inherit or make use of. Later in this chapter, we will see how these different classes are used in the framework.

- `XMLObject`. Classes inheriting from this class need to implement methods allowing to convert the object into an XML object or in an XML String or vice versa. This

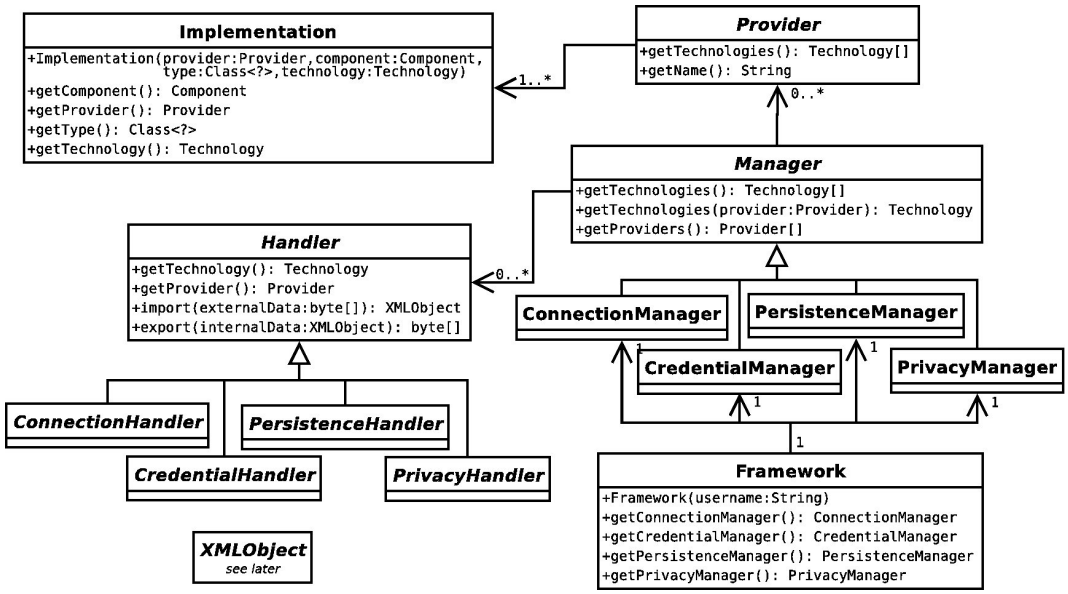

Figure 3.2: Class diagram for the framework.

facilitates sending framework objects over various types of connections and makes them programming language independent. Note that a `getXML()` or `getXMLString` may never disclose sensitive data such as secret keys (security) or attribute values (privacy) and that only the XML representation can be sent over connections. This way, the framework helps the application builder, since the accidential disclosure of sensitive data towards other parties is prevented; either the sensitive data is simply not shown in the XML representation (this is done in e.g. `Credential)`, or the class containing only sensitive data (e.g. `Attributes`) does not inherit from `XMLObject`, disabling the possibility to send the data to another party using the framework.

- `Parameters.` This abstract class inherits from `XMLObject`. Classes implementing this class will contain non-sensitive parameters such as the length of the modulus.
- `Data.` This abstract class does not inherit from `XMLObject` and can thus not be sent to another party using the framework. This will contain sensitive data such as attribute values or secret crypto data such as secret keys.
- `Technology.` A class that will be used by all handlers is the `Technology` class, which has a name and a version. It simply describes the applicable technology.
- `StatusInterface.` This interface is implemented by data objects that are the result of an interactive protocol with another party (e.g. authentication) and provides method signatures that allow to easily get hold of the returned data, as well as whether or not the protocol succeeded. If the protocol did not succeed, a status message must be provided.

Although it is not a part of this deliverable, it is possible to develop a uniform GUI for the framework usable by multiple applications, hence providing a consistent and uniform GUI.

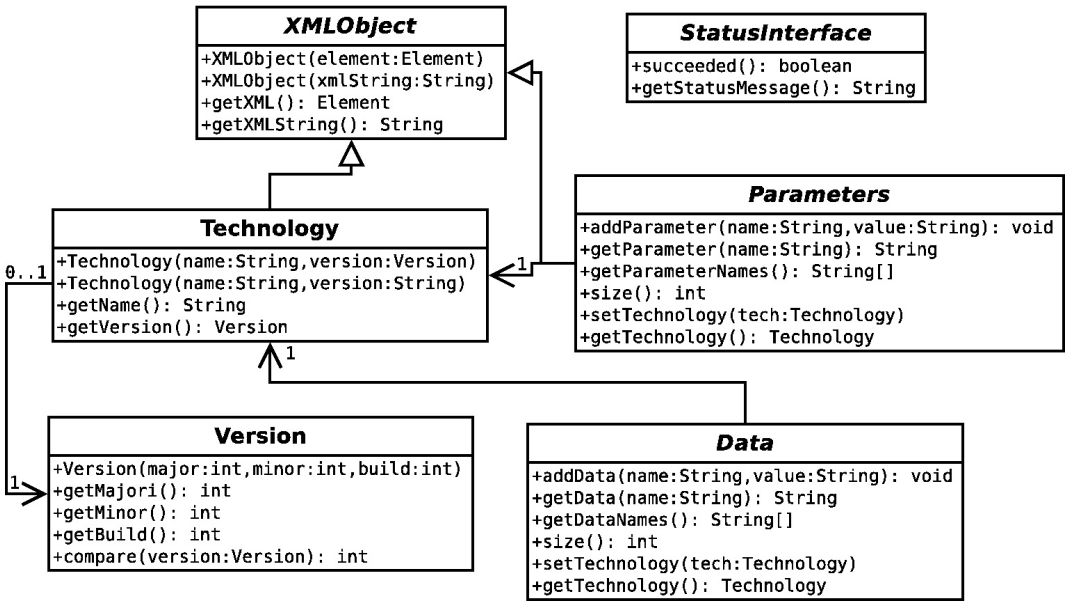

Figure 3.3: Classes related to the XMLObject class.

Secondly, a dispute manager with corresponding handlers, which could provide functionality to resolve conflicts, even if transactions where anonymous, is not considered in this deliverable. Generally speaking, we can state that a lot of effort has been devoted into the handlers and their implementations in providers. However, the managers are considered only on a higher, conceptual level in this deliverable.

3.1.2 Implementing a provider

In order to write a provider that can be plugged into the framework, the abstract `Provider` class needs to be implemented, as well as one or more of the `Handler` classes.

The Provider class

An example of a concrete `Provider` class is given in listing 3.1. Here, the provider provides three implementations for the credential handler and one for the connection handler. For each handler implementation, an `Implementation` object containing the required information must be added to the implementation of the abstract class `Provider`.

```
public class MyProvider implements be.kuleuven.cs.idf.Provider {

  @Override
  public Implementation[] getImplementations() {
    return new Implementation[] {
      new Implementation(this, Component.CredentialHandler,
          X509CredentialHandler.class,
          new Technology("X509", new Version("3"))),
      new Implementation(this, Component.CredentialHandler,
```

```
            BelgianEidCredentialHandler.class,
            new Technology("BelgianEid", new Version("3.5"))),
        new Implementation(this, Component.CredentialHandler,
            IdemixCredentialHandler.class,
            new Technology("Idemix", new Version("2.0"))),
        new Implementation(this, Component.ConnectionHandler,
            SocketConnectionHandler.class,
            new Technology("Sockets", new Version("1")))
        };
    }
```

Listing 3.1: Example of implemented provider class

A Handler class

The framework defines interfaces for different types of `Handlers`. To implement the generic interface for a specific `Handler` component, a class must be created that inherits the `Handler` component that will be implemented. An example of a `X509CredentialHandler` that implements the `CredentialHandler` interface is shown in listing 3.2

Note that the implementations in the providers will usually wrappers around existing implementations in order to allow these existing implementations to be plugged into the the framework.

```
import be.kuleuven.cs.idf.components.CredentialHandler;

public class X509CredentialHandler extends CredentialHandler {

  public X509CredentialHandler(Framework fw) {
    super(fw);
  }

  // implementation of CredentialHandler methods that are inherited
}
```

Listing 3.2: X.509 Credential Handler implementation

3.1.3 Local authentication to the framework.

Local authentication to the framework by the user is done as follows. Two handler threads are run; one client thread and one server thread. Communication between two threads happens using shared memory and standard synchronisation mechanisms (monitors). Hence, local authentication or remote authentication is conceptually the same. We will later see how a user can authenticate remotely. Based on the authenticatio, a secret is derived, giving the user access to his data.

3.2 Connection Handler & Manager

3.2.1 Handler Description

The connection handler offers a uniform interface to set up, listen for and break connections.

Each connections that leave the framework (e.g. to authenticate towards another entity or to retrieve a remotely stored credential) is handled by one of the appropriate `ConnectionHandler` objects. Multiple connections can be used simultaneously. The framework provides the same modules and interfaces for both clients and servers. However, they use complementary methods. Different technologies that can be implemented are (SSL over) TCP sockets, Bluetooth, NFC, anonymous networks like Tor and JAP, etc.

Figure 3.4 gives an overview of the classes related to the connection handler. The following classes are relevant:

- `ConnectionHandler.` This is the main class, used to set up, receive and break connections of a particular type. The class and all its methods are abstract and need to be implemented by the provider developer.

- `ConnectionParameters.` This class allows to set all the parameters relevant to set up or listen for a connection of a particular type. Each separate parameter has a name and an associated value.

  These parameters can be technology specific. For instance, if an SSL/TLS connection is set up, the `Credential` object (see section 3.4) required to do the authentication must be contained in the `ConnectionParameters` object, while such an object is not required to set up a TCP socket based connection. The list of required and optional parameters is technology specific and should therefore be well described by the provider developer in order to help the application developer. Section 3.2.4 describes how this can be done efficiently and intuitively for the application developer.

- `ConnectionListener.` A server can wait for an incoming connection. If this event occurs, a `connectionListener` is made, which allows to establish the incoming connection (using the `ConnectionHandler.accept()` method). The advantage of using a separate `ConnectionListener` object is that the server keeps listing for incoming connections while at the same time another connection is being established.

  This abstract class must be implemented by the provider developer.

- `Connection.` The actual connection which is returned by the `ConnectionHandler.connect()` method (client side) and by the corresponding `ConnectionHandler.accept()` method (server side). Objects of the following classes can be sent over a connection: `String`, `XMLObject`, `Request` and `Response`. Each connection has its own unique identifier. This will be useful later to reason about linkabilities of actions.

  The bookkeeping methods such as `getDuration()` are implemented by the framework.

- `Request.` A request contains a command (`String`) and additionally, extra data can be provided (`String`, `XMLObject` or `XMLObject[]`). For instance, `new Request( "GET_CREDENTIAL_TEMPLATE", "user")` could be a request for a credential template with the name "user". This class is fully implemented by the framework. The exact

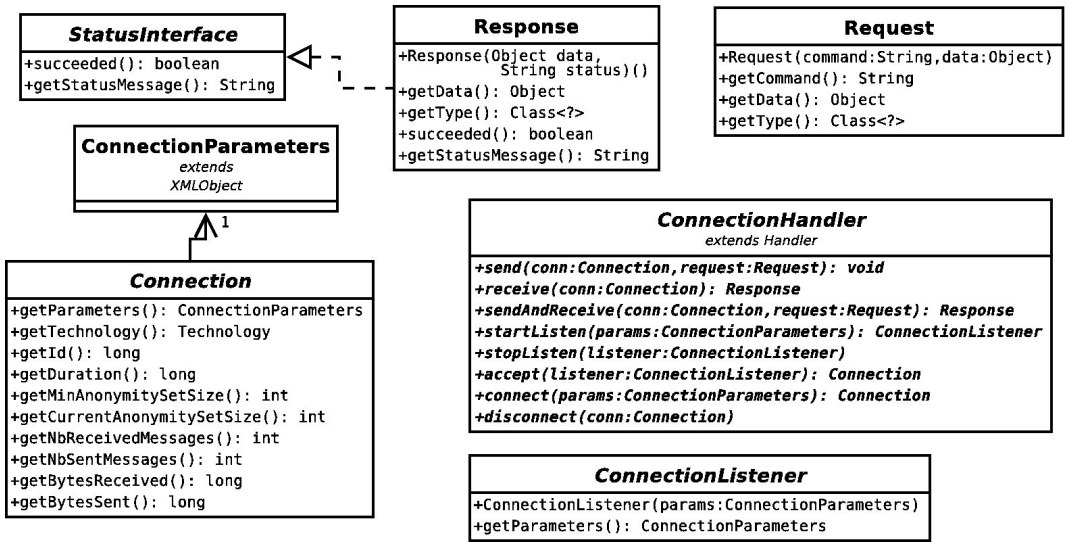

Figure 3.4: Class diagram for the connection handler related classes.

definition of the commands and the corresponding data is defined by the application developer.

- `Response`. After having sent a request, the other side replies with a `Response` object, which contains the returned data (can be empty) and a status message. The latter allows the sender of the `Request` object to get hold of the status; everything might have succeeded or an error message could be included as status message in the `Response` object. This class is fully implemented by the framework. But can be overriden by the provider developer (see section 3.2.4).

3.2.2 Implementation in a provider

The provider developer needs to implement `ConnectionHandler` and `ConnectionListener` completely. If a technology Abc is implemented, the class `AbcConnectionParameters` should be created, which inherits from `ConnectionParameters`. The methods in such a class contain a set of constructors which is used to set all the required and potentially the optional parameters. Also, setters can be added to set optional parameters. An example is given with parameters for a TCP sockets connection. This facilitates setting up the right parameters for a particular technology. Instead of writing

```
ConnectionParameters params = new ConnectionParameter();
params.addParameter("host", "server");
params.addParameter("port", 12345);
params.addParameter("timeout", 100);
```

to create a `ConnectionParameters` object appropriate for sockets, we can simply write

```
ConnectionParameters params = new SocketConnectionParameter("server", 12345,
    100);
```

which is less error-prone. A provider developer can easily extract the parameter values out of the `ConnectionParameters` object.

3.2.3 Connection handler usage example

In listing 3.3 an example is given of how an application at the client side could set up a connection and ask the server for a particular credential template (see section 3.4). The listing 3.4 lists the code of how a server could listen for an incoming connection and create the proper response.

```
// Get the connection manager
ConnectionManager connMgr = framework.getConnectionManager();

// Obtain proper connection handler
Technology tech = new Technology("Sockets", "1");
ConnectionHandler connHandler = connMgr.getConnectionHandler(tech);

// Establish connection.
ConnectionParameters params = new SocketConnectionParameters("198.112.23.78",
    12345);
Connection conn = connHandler.connect(params);

// Create and send request and receive response
Request request = new Request("GET_CREDENTIAL_TEMPLATE", "BeEId");
Response response = connHandler.sendAndReceive(conn, request);
if(response.succeeded())
  Template template = (Template)response.getData();
```

Listing 3.3: Example usage of connection handler by a client

```
// Get the connection manager
ConnectionManager connMgr = framework.getConnectionManager();

// Obtain proper connection handler
Technology tech = new Technology("Sockets", "1");
ConnectionHandler connHandler = connMgr.getConnectionHandler(tech);

// Listen for and accept incoming connection.
ConnectionParameters params = new SocketConnectionParameters("198.112.23.78",
    12345);
ConnectionListener connListener = connHandler.startListen(params);
while(true){
  Connection conn = connHandler.accept(connListener);
  // Server can start a new thread to process multiple incoming connections at
  // the same time.
  // Receive the request. Create and send the response
  Request request = connHandler.receive(conn);
  Response response;
  if(request.getCommand().equals("GET_CREDENTIAL_TEMPLATE")){
```

```
    String name = (String)request.getData();
    Template template = findCredTemplate(name); // app. specific method
    response = new Response("OK", template);
  } else {
    response = new Response("Unknown command");
  }
  connHandler.send(conn, response);
  connHandler.disconnect(conn);
}
```

Listing 3.4: Example usage of connection handler by a server

3.2.4 Application developer issues

The application developer has the possibility to inherit from the `Request` and `Response` classes in order to fine tune these to specific needs. For instance, if only integers are encrypted, a class `VencDecryptRequest` could be made as shown below, which is a specific request for a decryption of a verifiable encryption. An appropriate `VencDecryptResponse` could be made as well.

```
public class VencDecryptRequest extends Request{

  public VencDecryptRequest(int data){
    super("VENC_DECRYPT", data);
  }

  public int getData(){
    return (int)getData()
  }
}
```

Alternatively, a `VencDecryptor` class as shown below can be made. This class hides the communication issues for the user of the class. In the example code, it is assumed that all encrypted data are `String` objects. Once a `VencDecryptor` object `vd` is made and if a cipher `c` must be decrypted, we can simply call `vd.decrypt(c)`.

```
public class VencDecryptor{

  private ConnectionHandler handler
  private Connection conn;

  public VencDecryptor(ConnectionHandler handler, Connection conn){
    this.conn = conn;
    this.handler = handler;
  }

  public String decrypt(VerifiableEncryption cipher){
    Request request = new Request("VENC_DECRYPT", cipher);
    Response response = handler.sendAndReceive(conn, request);
```

```
    if(response.succeeded())
      return (String)response.getData();
    else
      return null;
  }
}
```

3.2.5 Manager Description

Depending on the communication requirements of the application (e.g. tamper proof, degree of anonymity, integrity- and/or confidentiality-protected, speed, etc.) the Communication Manager could select the most appropriate Communication Handler implementation. If the anonymity of a connection drops below a certain threshold, the connection could be broken, or the user could be warned.

3.2.6 Implementations

At the moment, implementations are available for:

- TCP sockets
- HTTP servlets

3.3 Persistence Handler & Manager

3.3.1 Handler Description

A persistence handler is responsible for the storage of data objects (`XMLObject`). This could be locally on a hard drive or a USB stick, but as well on a smart card with access control or even remotely. An encryption mechanism can be present as well. Figure 3.5 shows the class diagram. The following classes are relevant at the handler level:

- `PersistenceHandler.` An implementation of this abstract class provides the functionality to store, load and delete data objects (`XMLObject`) using a particular persistence technology. Examples of such technologies are persistence using a particular type of smart card and persistence on remote server using a particular protocol. It allows to retrieve all the names of object of a particulaar type stored using a particular `PersistenceParameters` object.

  All the methods in this class are abstract and must be implemented by the provider developer.

- `PersistenceParameters.` Since, in order to keep them stateless, persistence handlers only contain the functionality of how persistence related operation should be done and not where and with what parameters. This information is contained in `PersistenceParameters` objects. Besides location information, also `Credential` and `ShowSpecification` objects can be part of a `PersistenceParameters` object, as well as tokens (e.g. a password or a symmetric key) required to access the data.

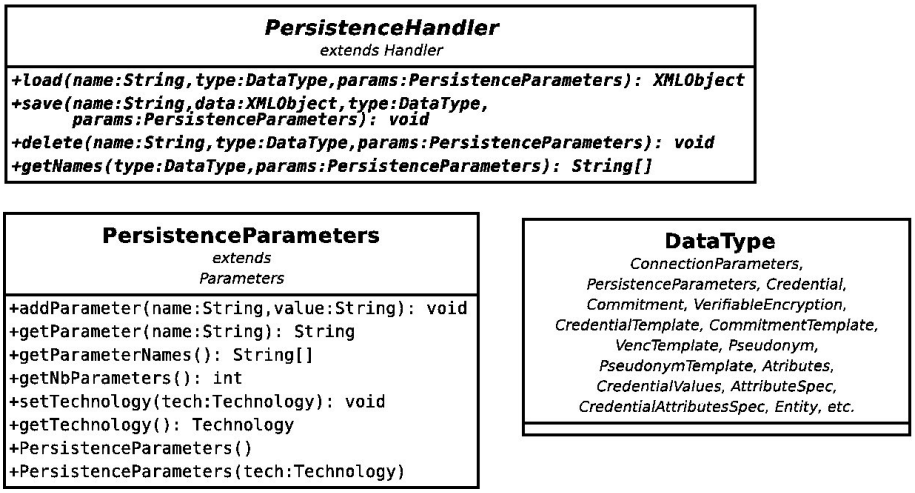

Figure 3.5: Class diagram for the persistence handler.

- `DataType`. This is an enumeration of all the data types that can be made persistent in the framework.

3.3.2 Implementation in a provider

The provider developer needs to implement `PersistenceHandler` completely. If a technology Abc is implemented, the class `AbcPersistenceParameters` should be created, which inherits from `PersistenceParameters`. The methods in such a class contains a set of constructors which is used to set all the required and potentially the optional parameters. Also, setters can be added to set optional parameters. This is the same technique that is applied for `ConnectionParameters` in the previous section.

At the moment, only local stoarage as XML file is provided.

3.3.3 Usage by application developer

```
// get the manager
PersistenceManager pMgr = fw.getPersistenceManager();

// The data we want to store
XMLObject data = template;

// Get the persistence handler
Technology tech = new Technology("XMLFile", "0.0.1");
PersistenceHandler pHandler = mgr.getPersistenceHandler(tech);

// ... and set the corresponding PersistenceParameters
PeristenceParameters params = new XMLPersistenceParameters("store.xml");
```

```
// Save the template
pHandler.saveTemplate(data, params);

// Load a credential data object using the same handler and parameters.
// Credential inherits from XMLObject.
Credential cred = pHandler.load("drivingLicense", DataType.credential, params);

// Delete that credential
pHandler.delete("drivingLicense", DataType.credential, params)
```

3.3.4 Manager Description

The manager keeps track of the following: 1) the available persistance handlers, 2) the available persistence parameters and 3) for each stored data object, its name (identifier) and a reference to the proper `PersistenceHandler` and `PersistenceParameters` object. The persistence manager offers the functionality to query for stored data objects that have particular properties.

Secondly, the manager could build on top of the persistence parameteres and handlers places. Places are logical storage entities that make abstraction of the underlying technology. A user could for instance have a 'wallet', containing only his most important credentials and a number of receipts, although the content of the wallet may reside on different stores or might be duplicated on several stores.

3.4 Credential Handler & Manager

The most important interface in the framework is the `CredentialHandler` interface. It defines a generic interface to perform actions using credentials (signing, authenticating, issuing credentials, etc.) while making abstraction of a specific technology (X.509, Idemix, etc.). This interface can be implemented by several providers using different technologies. Each `CredentialHandler` implementation (implemented in a provider) can only handle one type of credential (e.g. `X509CertificateCredentialHandler`, `PseudonymCertificateCredentialHandler`, `IdemixCredentialHandler`, `BelgianEidCredentialHandler`). The `CredentialManager` manages and selects the most appropriate credential(s) in the framework.

In order to explain the `CredentialHandler` interface, first, some other concepts and their related class diagram need to be explained. In section 3.4.1, the class diagram for pseudonyms, commitments, verifiable encryptions and credentials is explained. Section 3.4.2 explains the templates and their relevant classes. Section 3.4.3 elaborates on the shoz specification, which describes the proeprties to be disclosed. Section 3.4.4 introduces the concept of disclosures, which is a show specification with the requried other objects in order to do a verification or proof. Section 3.4.5 introduces the concept of an entity and section 3.4.6 introduces the transcripts, which are the result of a protocol. Section 3.4.7 introduces attriubte values, which are needed to create credentials. In section 3.4.8, the actual credential handler is introduced. Section 3.4.9 explains how commitment and credential templates cna be made and section 3.4.10 continues by explaining how commitments and verifiable encryptions can be created and used. Finally, section 3.4.11 briefly touches the credential manager.

3.4.1 Credential, Pseudonym, Commitment and VerifiableEncryption

In figure 3.6, the classes related to commitments, verifiable encryptions, pseudonyms and credentials are shown. These concepts have in common that properties or at least ownership of it can be proven or verified.

- `ProverObject`. A concrete class (inderectly) inheriting from this abstract class is a `Credential`, a `Commitment`, a `Pseudonym` or a `VerifiablEncryption`. Concrete `AttributeProverObject` objects all contain attributes about which properties can be proven. `AttributeContainer` objects are more simple than `Credential` objects in the sense that their attributes are not certified by a third party. Each `ProverObject` has a reference to a template (see 3.4.2) and to a `Data` object. The former describes the type of the commitment, pseudonym, credential or verifiable encryption, while the latter contains secret prover data that must not be disclosed to others. Therefore, `Data` does not inherit from `XMLObject` and these data are not shown in the XML representation of the `ProverObject` objects. `AttributeProverObject` objects also have a reference to an `Attributes` object. The XML representation also hides the exact attribute values of the `AttributeProverObject` objects. Sending `ProverObject` objects to another party hence does not disclose any sensitive information. Since the `ProverObject.getData()` method is protected, this data is only visible within the framework. The final type of `ProverObject` is the `Pseudonym`, which can also have secret data.

- `Attributes`. An instantiation of this class contains zero, one or more `Attribute` objects. Each `Attribute` object contains a type, a name and a value. `CredentialAttribute` objects in addition contains information about who did provide the information (the issuer, the receiver, or the issuer using a commitment). In the latter case, the corresponding `CommitmentTemplate` must be provided. Note that none of the classes related to `Attribute` inherits from `XMLObject` since these objects do not need to be sent to other parties.

Attribute containers and pseudonyms can either contain the secret (`CommitmentData`, `VencData` or `PseudonymData`) required to prove properties about the content or these data might be removed. In the latter case, the objects can only be used to verify and not to prove properties. Additionally, in case of commitments or verifiable encryptions, the `AttributeValues` object can be removed (i.e. the sensitive data). These data are removed by the framework if the object is sent to another party, or if the `removeProveCapabilities()` method is called and are not shown by the `getXML` methods.

The different `ProverObject` classes and their `Data` classes need to be implemented by the provider developer. In the next section, the `templates` are considered.

3.4.2 Template

In figure 3.7, the main `Template` class and its related classes are shown. The main classes are now explained.

- `Template`. Abstract class from which the different concrete template classes (`CommitmentTemplate`, `VencTemplate`, `PseudonymTemplate` and `CredentialTemplate`) inherit. A template has a technology and a set of securityparameters (determining for instance the sizes of the keys). A `CredentialTemplate`

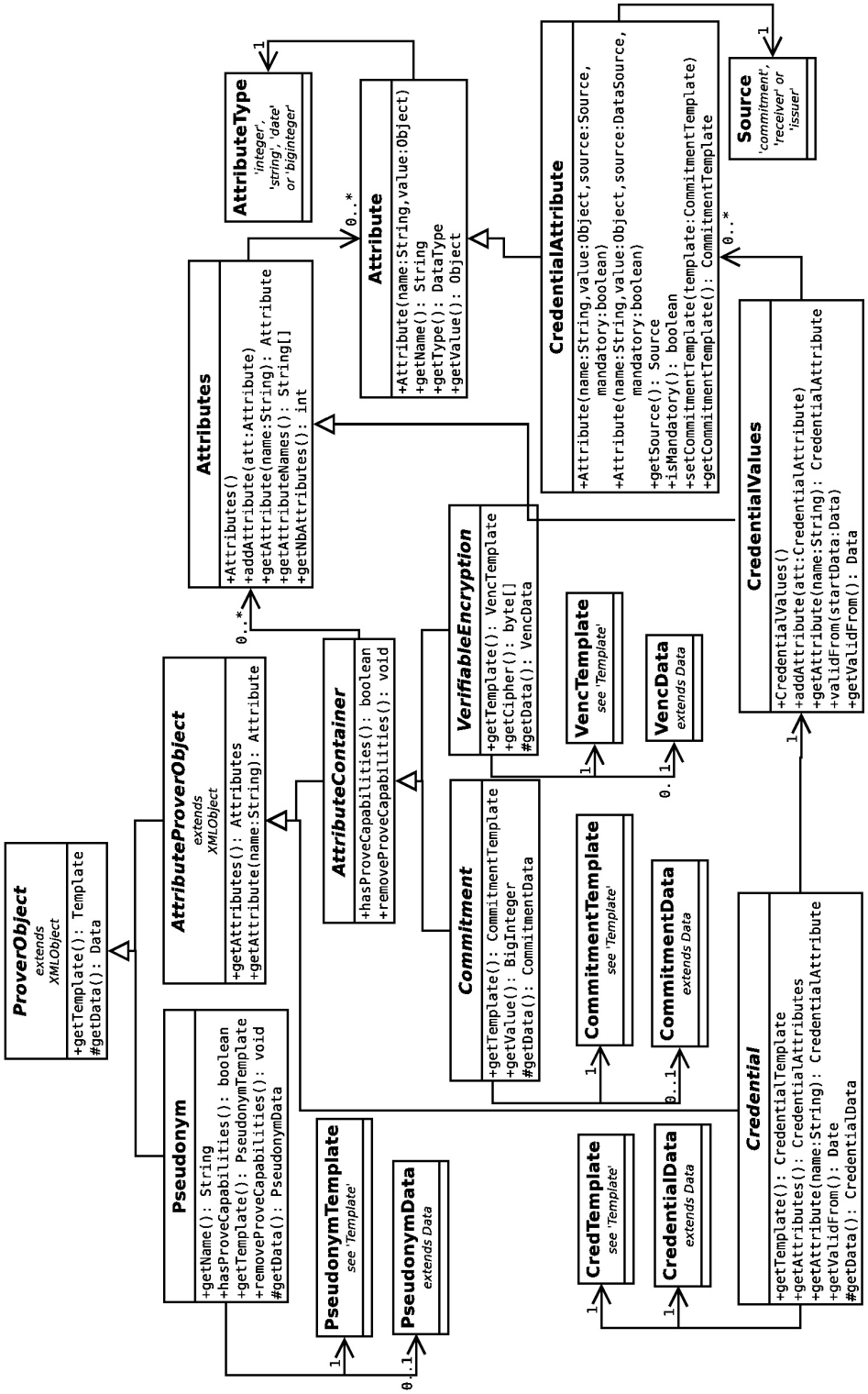

Figure 3.6: Classes related to Credential, Pseudonym, Commitment and VerifiableEncryption.

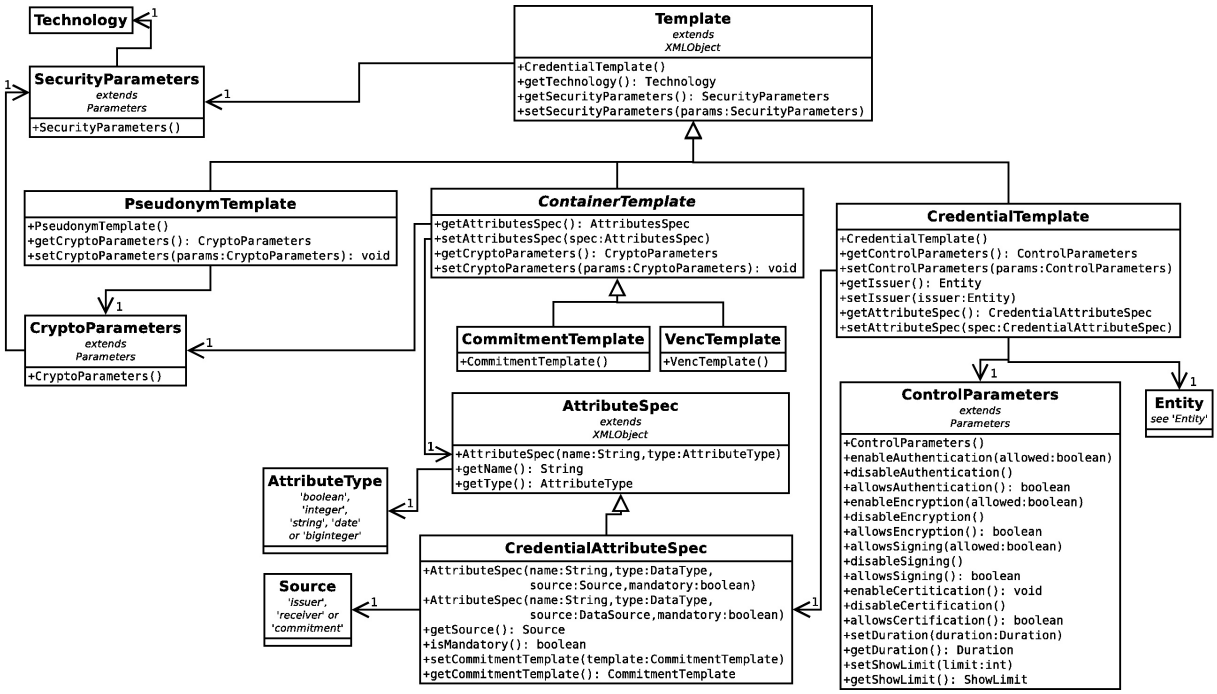

Figure 3.7: Class diagram for templates.

has more complex attribute specifications (`CredentialAttributeSpecification`), it has a `ControlParameters`, defining how the credential can be used and a credential template has an issuer, which is an `Entity` object (see section 3.4.5).

- `AttributeSpec.` `CommitmentTemplate` and `CerifiableEncryptionTemplate` objects have a specification of the attributes that must be integrated in the template. `CredentialTemplate` objects have more complex attribute specifications and `pseudonyms` have none.

- `CredentialAttributeSpec.` Since credentials are more versatile than commitments and verifiable encryptions, the attribute specification for credentials is more elaborate. It contains the information about how the attribute value should be provided (`Source`). either by the issuer, by the verifier, or the attribute value could be provided by the receiver but in a commitment. In the latter case, a `CommitmentTemplate` should be given as basis to generate that commitment. The credential attribute specification also defines that the attributes is either mandatory or obligatory.

- `ControlParameters.` An object of this class describes how, when and how many times the credential corresponding to a credential template can be used. In addition, it contains information about how to check the validity (e.g. the link to the CRL).

The provider developer needs to define the content of the `SecurityParameters`. Usually, the application developer will put the logic to create `CryptoParameters` objects in the appropriate handler.

3.4.3 ShowSpecification

In figure 3.8, the `ShowSpecification` class and other related classes are depicted. A show specification is a description of the properties that will be or are disclosed, however, it does not containe these objects.

- `ShowSpecification.` An object of this class describes exactly what is (going to be) disclosed as part of the authentication or signature protocol. It defines the predicates (e.g. $cred.DoB > 1980$) and the nym under which the properties are or must be shown. It can also contain the name of the nym under which the authentication/signing will be done.

- `Predicate.` A predicate is either a `BasicPredicate`, an `IntervalPredicate`, a `Connection` or a `PossessionPredicate` object. A `BasicPredicate` is a relation between two terms (see next bullet), while a `Connection` is the union or intersection of two or more `Predicate` objects. A `PossessionPredicate` predicate proves possession of a credential and an `IntervalPredicate` proves that a value in a credential, commitment or verifiable encryption lies between two other values.

- `Term.` A term object is either a constant (`ConstantTerm`), a reference to an attribute in a credential, commitment or verifiable encryption (`CredentialTerm`, `CommitmentTerm` or `VencTerm`), or a composite term (`CompositeTerm`), which is the result of applying an operator one two or more other `Term` objects.

Note that the pseudonyms, credential templates, commitments and verifiable encryptions as such ar not part of the show specification. These can either be directly given to the methods in the `CredentialHandler` instantiation, or can be put in a `Disclosure` object (see section 3.4.4), which can as well be given as parameter to the a `CredentialHandler` object.

Also note that the show specification defines the minimum that will be or is disclosed by the prover during an authentication or a signature. Depending on the used technology, it is possible that more is disclosed. For instance, proving possession of a X.509 certificate implies disclosing all the attributes contained in the certificate.

The provider developer does not need to implement the conversion of a showspecification to the particular technology.

3.4.4 Disclosure

A `Disclosure` object contains all the information needed to either authenticate or sign, or to verify an authentication or signature. The `Disclosure` class and its related classes are shown in figure 3.4.4.

- `Disclosure.` A `ShowSpecification` is only a description of what is disclosed or what will be dislcosed. However, in order to do the actual authentication or place the actual signature, also the `Commitment`, `Pseudonym`, `Credential` and `VerifiableEncryption` objects must be provided by the prover. These objects are together with the `ShowSpecification` put in a `Disclosure` object. In the case of the verifier, no `Credential` objects are given, but the corresponding `CredentialTemplate` objects and, evidently,the sensitive data of the different prover objects has been removed. By calling the method `isValidProvableDisclosure`, one can check

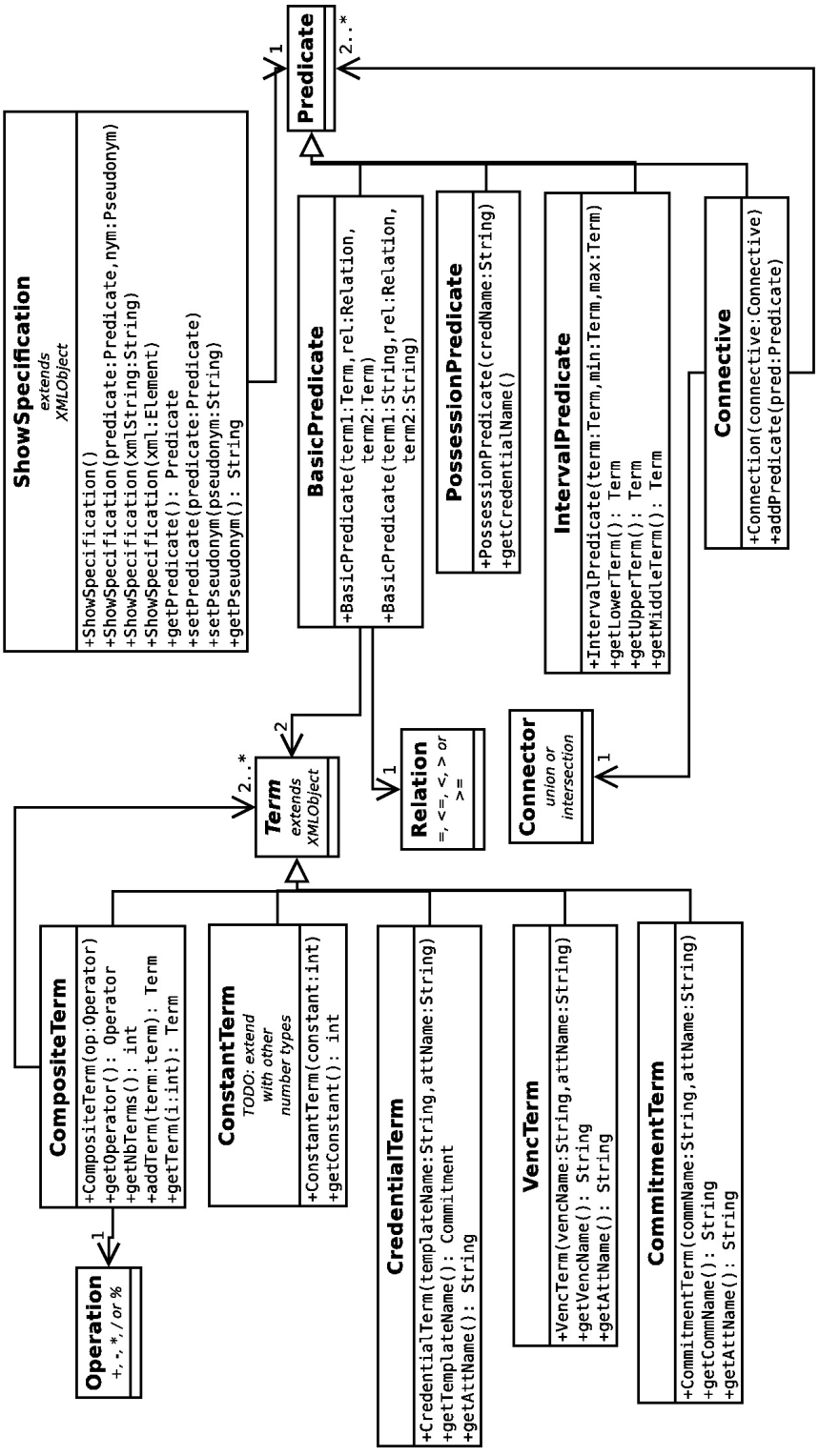

Figure 3.8: Class diagram the show specification.

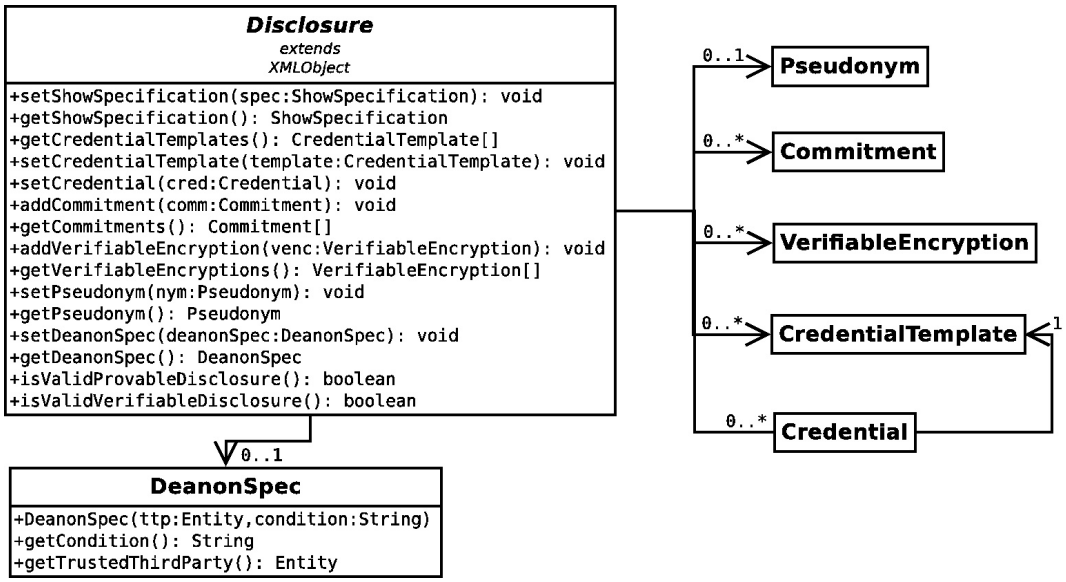

Figure 3.9: Class diagram for classes related to `Disclosure`.

whether the disclosure suffices to prove the `ShowSpecification` object. The method `isValidVerififableDisclosure` returns whether the disclosure suffices to verify the properties stated in the show specification.

- `DeanonSpec`. Objects of this class can be added to a `Disclosure` object and contain the condition under which deanonymization of the authenticator or signer is allowed and by whom. Multiple `DeanonSpec` objects can be added.

Note that the commitments, pseudonyms and verifiable encryptions will have their secret data (`Data`) removed when sent over a connection. Also, the `Attributes` object of commitments and verifiable encryptions will be removed in that case. Finally, if credentials are part of the disclosure, they will be removed and only their template will be kept. This corresponds to the information returened by a `getXML()` method.

The provider developer does not need to implement anything w.r.t. `Disclosure` objects.

3.4.5 Entity

An entity is a set of personal information that is disclosed during a single authentication or signature. It can have a proof, which implies that another entity guarantees the correctness of the information. Entity objects are useful to verify certification chains (or variants) and to keep track of the information that is known to or about others.

The main classes are:

- `Entity`. A set of disclosed data (in `VerifiableDisclosure`), potentially associated with a proof (`DisclosureProof`). Such a proof can either be an `Authentication` or a `Signature` object.

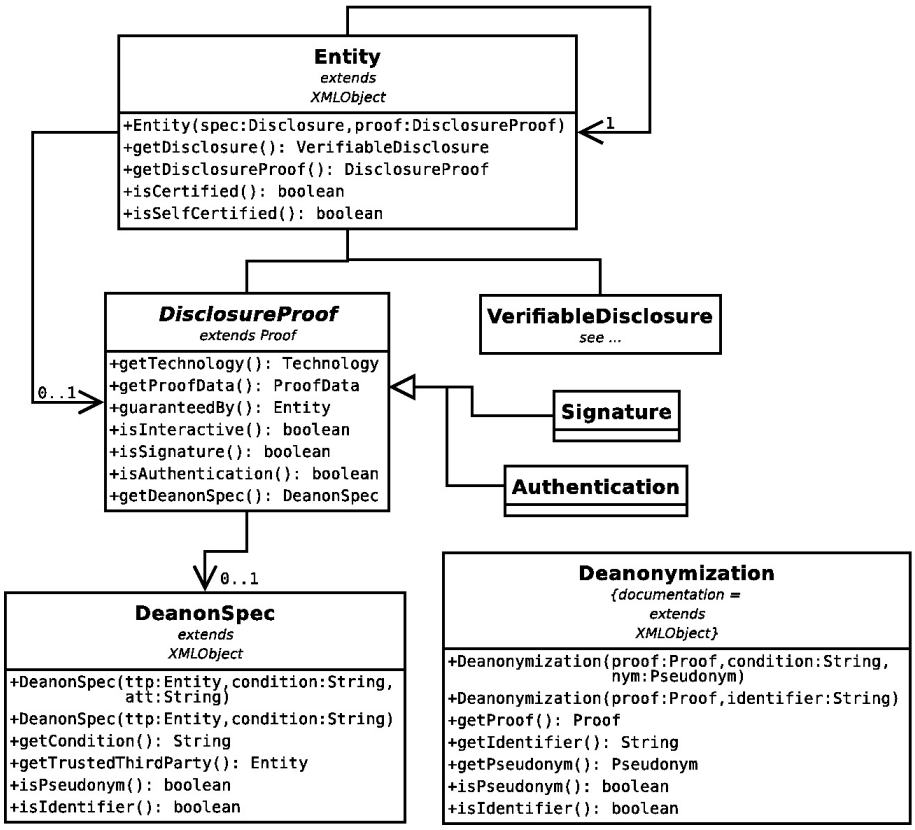

Figure 3.10: Class diagram for predicates in the credential handler.

- `DeanonSpec`. Such an object can be part of the proof. Since it is bound to the proof, the deanonymizer is sure that he deanonymizes the right identifier or pseudonym.
- `Deanonymization`. The result of a denonymization. It cqn containt a proof of the correcness of the deanonymization.

For instance, signing a document using an X.509 certififcate will result in a `Signature` object, which will also contain (parts of) the X.509 certificate. This combination is a proof for the `Disclosure` object, which will contain all the disclosed attributes. In case of an Idemix authentication, the resulting proof will be part of the `DislosureProof` object, and the `Disclosure` object used during the authentication is refenced to by the `Entity` object.

The provider developer needs to ensure that the proper data is put into the objects.

3.4.6 Transcript

For each of the involved parties, an interactive protocol results in a transcript. This transcript contains all relevant information known to that particular party. An overview is given in figure 3.11.

- `Transcript`. Each `Transcript` object contains at least a timestamp of the start of the protocol and one of the protocol end. Also, data about the connection is con-

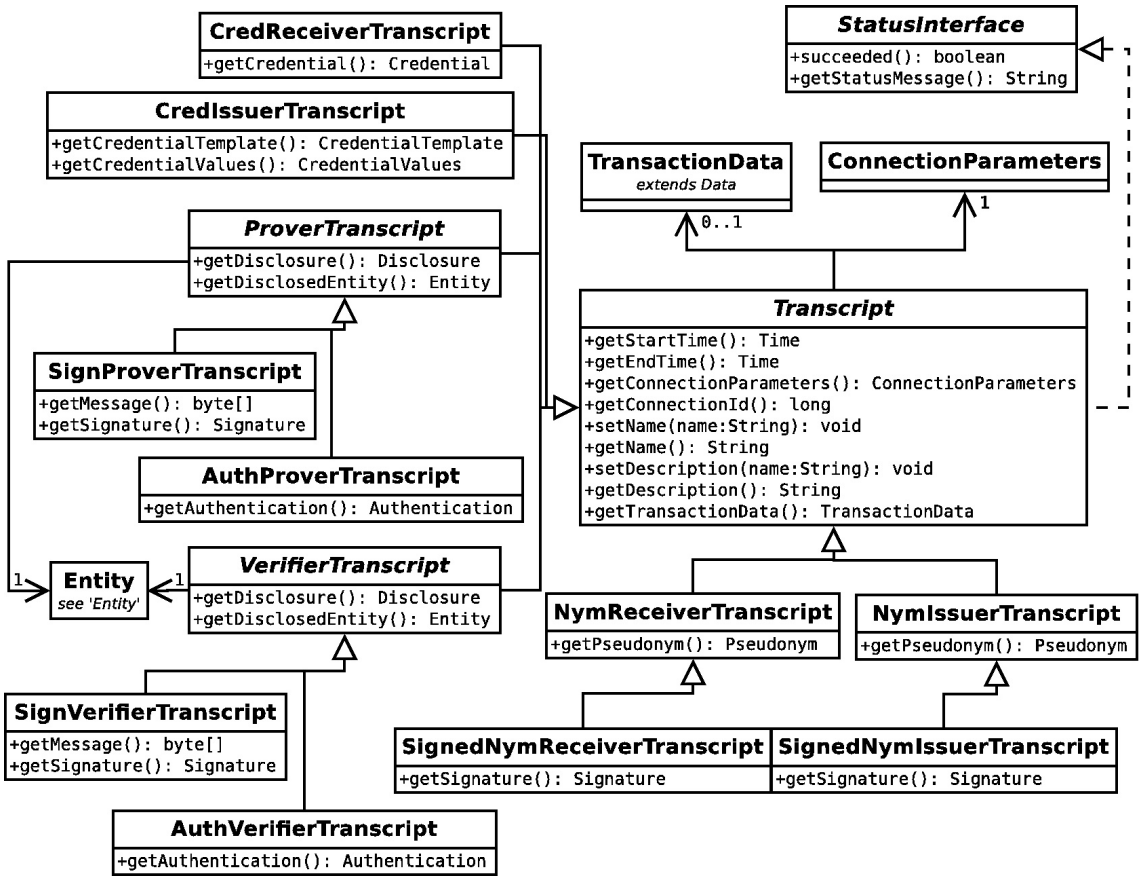

Figure 3.11: Class diagram for transcripts and entities.

tained in a transcript; the connection parameters and the connection identifier. For his own use, the transcript owner or the application can give the transcript a name and description, which might be useful to retrieve it later, for instance in case of liability issues. A `Transcript` object can also contain data about the transaction iteself (`TransactionData`). An `IssuerTranscript` object contains additionally the template of the credential that has been issued. A `ReceiverTranscript` object contains the newly received credential. `ProverTranscript` and `VerifierTranscript` objects also contains a reference to an `Entity` object (see section 3.4.5). The pseudonym related transcripts contain a pseudonym; the receiver is able to prove possession of it, while the issuer is not. In case of signed nym transcripts, the signature is contained in the transcript.

Note that the `ShowSpecification` in the a `Disclosure` object in a transcript describes the information that has actually been disclosed. This might be more than the description of the `ShowSpecification` object that was used during the signing or authentication protocol. For instance, proving possession of an X.509 certificate implies disclosing all contained attributes.

The provider developer can optionally set the transactiondata.

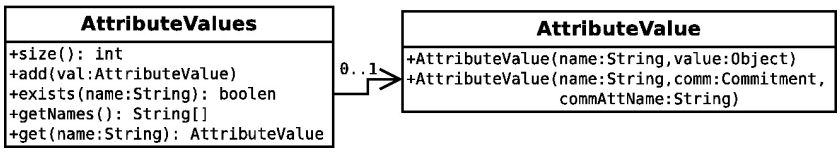

Figure 3.12: Class diagram for attribute values.

3.4.7 AttributeValues

In order to do a credential issuance, both the receiver and the issuer need to specify zero or more attributes to be included in the credential; These values are described in two `AttribueteValues` objects; one at the issuer side and one at the receiver side. The attributes given a value can differ in both objects. An `AttributeValues` object at the receiver's side can even contain a reference to a commitment about which properties were proven to the issuer beforehand. The two relevant classes are shown in 3.12.

3.4.8 Credential Handler

The credential handler offers the interface to use the concepts described earlier in this document. The interface is given in figure 3.13. The typical methods are now explained.

- `authenticate().` A party who wants to authenticate to another party calls this method. If there are only a very limited number of other parameters, they can be given as separate parameters, Otherwise, the a `Disclosure` parameter (with prove capabilities) must be given as second parameter besides the connection. The authentication protocol results in a `AuthProverTranscript` (see 3.4.6).

- `receiveAuthenticate().` This method is complementary to the previous one and is called by the verifier. Instead of a `Credential` object, a `CredentialTemplate` object is given as parameter, and the `Disclosure` object may not have prove capabilities, since only properties need to be verified.

- `sign().` This method is called in order to sign a message, which is given as a parameter. If a `Connection` object is given as first parameter, the signature is interactive, and has hence only proof value towards the verifier. If no connection is given, the signature is generated locally and could be sent to and verified by anyone afterwards. The last parameter is a `Disclosure` object (with prove capabilities) which describes the properties that are linked to the signature. Instead of this `Disclosure` object, a credential or a credential and a show specification could be given as parameters. Note that the pseudonym, commitments and verifiable encryptions in the disclosure object must have prove capabilities.

- `verifySign().` This method is complementary to the previous one. If a connection is given as parameter, it is an interactive signature, if not it is a non-interactive signature. Instead of credentials, credential templates are given as parameters or put in the `Disclosure` object. Only an interactive signature results at each side in a transcript conainting the signature. Otherwise, the signature itself is returned by the `sign()` function or a boolean is returned by the `verifySign()` function.

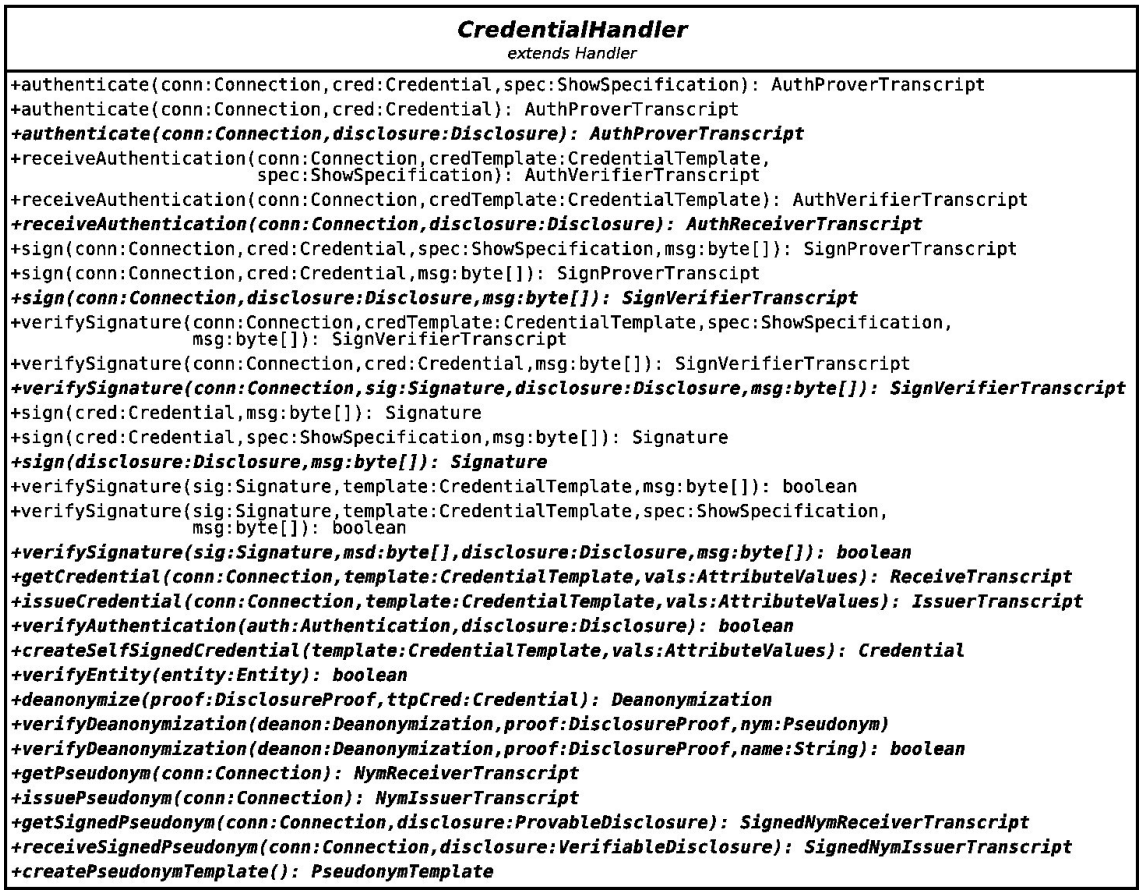

Figure 3.13: The credential handler interface.

- `getCredential()`. This method is called in order to receive a credential. A connection, a credential template and a `CredentialValues` object are given as parameters. The latter determines the values that may be chosen by the receiver of the credential. The other values must be chosen by the issuer. A `CredentialValues` object can also contain references to attributes in commitments, about which properties might have been proven beforehand by the receiver to the issuer. These values are included in the credential without the issuer knowing them.

- `issueCredential()`. This is the method complementary to the previous one, in which the issuer provides in an `AttributeValues` object the attribute values that may not be chosen by the receiver.

- `verifyAuthentication()`. An authentication can be verified afterwards using this function. The `Authentication` and `Disclosure` object must be given as parameters and a boolean is returned.

- `createSelfSignedCredential()`. This function allows to generate a self signed credential based on a credential template and the corresponding attribute values which are given as parameters to the function.

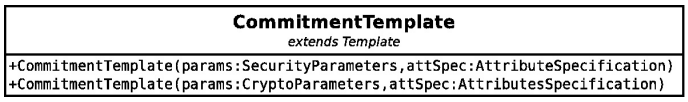

Figure 3.14: Interface to create commitment templates.

- `verifyEntity()`. This function is useful to verify a certification chain or other directed certification graphs; Each authentication or signature is guaranteed by certifier, which might in turn be certified by yet another entity. The correctness of all the entitites in this directed graph can be verified using this method.

- `deanonymize()`. Based on an `DisclosureProof` object (extracted out of a transcript) and the `DeanonSpec` object, a deanonymization can be done by the trusted third party. The result of this method returns a `Deanonymization` object.

- `verifyDeanonymization()`. The `Deanonymization` object returned by the previous method can contain a proof that the content indeed corresponds to what the TTP claims. This method allows to verify this proof.

- `getPseudonym() & issuePseudonym()`. These methods are complementary methods to issue a pseudonym by a service provider to a client. Analogous, there are the `getSignedPseudonym()` and the `issueSignedPseudonym()` methods to which an extra `Disclosure` object is given as parameter resulting in a provable binding between the pseudonym and the disclosure by the issuer.

- `getPseudonymTemplate`. Based on some security parameters, a pseudonym template can be made.

The provider developer needs to implement all or a subset of the abstract methods in this handler.

3.4.9 Commitment and Verifiable Encryption creation

Figure 3.14 shows the interface provided by the `CommitmentTemplate` class. A specification of the attributes must be given to both constructors. The first parameter can be chosen; either it is a `SecurityParameters` object or a `CryptoParameters` object. In the first case, the crypto parameters such as the modulus and bases will be generated during construction of the template. In the latter case, this is not necessary since they are given as parameter.

Figure 3.15 shows the different relevant classes for verifiable encryptions.

- `VencKey`. A `VencKey` object is either a public verifiable encryption key (`VencPublicKey`) or a private verifiable encryption key (`VencPrivateKey`). A `VenKey` has a reference to a `CryptoParameter` object. A `VencKeyPair` is a public-private pair of such keys.

- `VencTemplate`. This objects inherits from `ContainerTemplate`. In order to create a template, the specification of the attributes must be given. The other parameter is a verifiable public key.

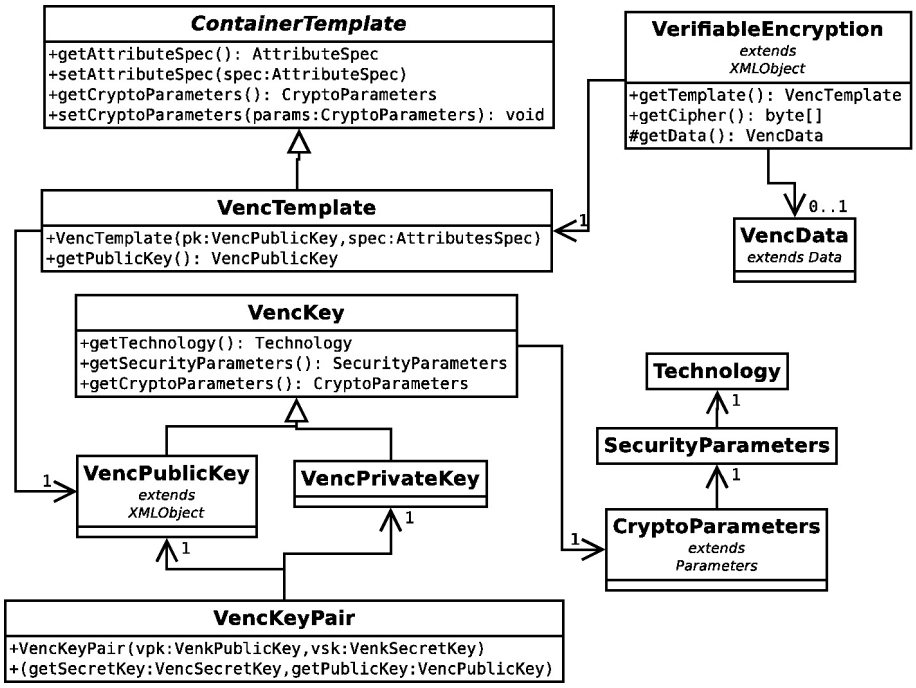

Figure 3.15: Verifiable Encryption related classes

- `VerifiableEncryption`. An object of this class is a verifiable encryption of attribute values corresponding to the `AttributeSpecification` object in the template. The `VencData` object does not leave the framework and allows to prove properties about the encrypted content. If the verifiable encryption is sent over a connection, this data is removed and hence the ability to prove properties about the encrypted attributes. The data is not shown by a `getXML()` method call. The cipher itself is a byte array.

3.4.10 Using comitments and verifiable encryptions

Figure 3.16 shows that two classes inherit from `AttributeContainerHandler`; `CommitmentHandler` and `VencHandler`. The former allows to create commitments, while the latter allows to create verifiable encryptions and to decrypt them. Proving properties about attributes in verifiable encryptions or commitments can be done using the `prove()` methods provided by the `AttributeContainerHandler`. These methods can be interactive or non-interactive. Only in the former case, a connection must be given as parameter. `AttributeContainerHandler` also provides the corresponding `verify()` methods.

Based on a commitmentvalue and attribute values, a commitment can be made. Again, the commitment will contain a `CommitmentData` object allowing to prove properties, however, this object never leaves the framework of the creator. A `VencHandler` object allows to create verifiable encryption key pairs, to encrypt and to decrypt verifiable encryptions.

The provider developer needs to implement all or a subset of the abstract methods in this handlers.

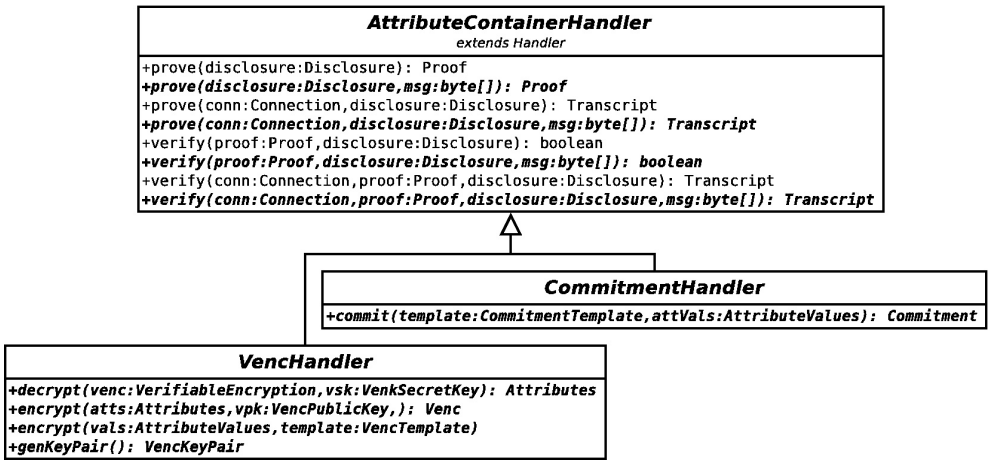

Figure 3.16: Verifiable encryption handler and commitment handler

For the credential handlers, wrappers have been writtten around two versions of Idemix, around a bouncycastle implementation of X.509 certificates. Based on this, pseudonym certificate support has been integrated and finally, support for the Belgian eID card using the official middleware provided by the Belgian governement has been integrated.

3.4.11 Credential Manager

The credential manager allows the service provider to compose an authentication request or a signature request.

This request contains a list of templates that are acceptable by the service provider. Hence, the client is only allowed to use credentials that have one of the listed credential templates. These templates contain the acceptable technologies, security parameters and trusted issuers. Potentially, a template can also contain information about the way the correctness of the contained attributes has been verified by the issuer.

This request can define the minimum and optional properties that must or may be disclosed in order to get access to a particular service offered by the service provider. For instance a client must be older than 18 and living in an area with ZIP code 3000. Additionally he can disclose that (s)he is in the age interval [18-35] or that (s)he is older than 35.

If a message needs to be signed, the structure of the message can be defined in the request. For instance, in an ePoll, the client might be given the choice between different options (e.g. in favour or against). This choice might be concatenated with the description and identifier of the poll and the current time.

Also, the deanonymization condition and a list of trusted deanonymizers can be given in the request.

The credential manager of the client will receive the request from the service provider and will find all the sets of credentials, commitmentes, etc. that can fullfill the service provider request. The framework will, based on its policy, with the help of the privacy manager and potentially with user intervention select the credentials, properties and message that will be disclosed and how. This is called the client's selection.

Based on this the service provider request and the client's selection, the authentication or signature process can be performed. The credential managers at both sides will select the appropriate handlers during this process.

3.5 Credential handler interface usage examples

The usage of the credential handler and the related classes is now illustraded.

3.5.1 Commitment creation

This section explains how `CommitmentTemplate` and `Commitment` objects can be created. We repeat that a `CommitmentTemplate` object consists of a technology, security parameters (length of the modulus, order, etc.) and crypto parameters (modulus, bases, etc.). The security parameters must be derivable from the crypto parameters and the technology must be derivable from the security parameters.

If the crypto parameters are known beforehand, the approach in listing 3.5 can be adopted. In this example, the crypto values are simply extracted from another (IdemixCredential) object. In 3.6, only the security parameters are given, which means that the commitment handler will have to find/calculate new crypto values, which might take some time.

The `CommitmentTemplate` object contains the technology, security parameters and optionally the crypto parameters. The attribute specification is given as argument, together with either the crypto or the security parameters.

```
IdemixCredential cred = ...
CommitmentTemplate cTemplate = new CommitmentTemplate();
CryptoParameters params = new PedersenCryptoParameters(cred)
AttributesSpec spec = new AttributeSpecs();
spec.add("name", AttributeType.String);
spec.add("gender", AttributeType.Integer);
cTemplate.setCryptoParameters(params);
cTemplate.setAttributesSpec(spec);
```

Listing 3.5: Creation of a Pedersen commitment template based on crypto parameters in an Idemix Credential

```
CommitmentTemplate cTemplate = new CommitmentTemplate();
SecurityParameters params = new PedersenSecurityParameters(1024, 128);
AttributesSpec spec = new AttributeSpecs();
spec.add("name", AttributeType.String);
spec.add("gender", AttributeType.char);
cTemplate.setSecurityParameters(params);
cTemplate.setAttributesSpec(spec);
```

Listing 3.6: Creation of a Pedersen commitment template based on the security parameters (such as length of the modulus)

A commitment creation is shown in listing 3.7. Based on a commitment template and the attribute values, the commitment is generated.

```
CommitmentHandler commHandler =  credMgr.getCommitmentHandler("Pedersen", "
    0.0.1")
AttributeValues vals = new AttributeValues();
vals.add('name', "Kristof Verslype');
vals.add('gender', 'm');
Commitment comm = commHandler.createCommitment(template, vals);
```

Listing 3.7: Creating a commitment. If only security parameters are given

3.5.2 Verifiable encryption creation.

Listing 3.8 illustrates how a verifiable encryption template can be made and listing 3.9 shows how a verifiable encryption can be generated.

```
VencPublicKey pk = ...
AttributesSpec spec = new AttributeSpecs();
spec.add("name", AttributeType.String);
spec.add("gender", AttributeType.Integer);
VencTemplate vTemplate = new vTemplate(pk, spec);
```

Listing 3.8: Creating of a verifiable encryption template

```
VencHandler vencHandler =  credMgr.getVencHandler("Camenisch", "0.0.1")
VencTemplate vTemplate = ...
AttributeValues vals = new AttributeValues();
vals.add('name', "Kristof Verslype');
vals.add('gendere', true);
VerifiableEncryption venc = vencHandler.createVenc(vals, vtemplate);
```

Listing 3.9: Creating of a verifiable encryption

3.5.3 Create a self signed X.509 certificate.

A root certification authority will need a self-signed credential. This is illustrated in listing 3.10

```
CredentialManager cMgr = ....
CredentialHandler credHandler = mgr.getHandler("X.509", "0.0.1");

// Create the credential template. we start by creating the security parameters
SecurityParameters secParams = new X509SecurityParameters(1024, "SHA1withRSA");

// Since the credential is self signed, no entitiy needs to be set as issuer.
// The control parametes are set
ControlParameters conParams = new ControlParameters();
conParams.disableEncryption();
conParams.disableAutentication();
conParams.disableSigning();
```

```
conParams.enableCertification();
conParams.setDuration("2 years");

// The attributes specification is set
CredentialAttributeSpecs attSpecs = new CredentialAttributeSpecs();
attSpecs.add("name", AttributeType.String);
attSpecs.add("address", AttributeType.String);

// Finally, the pieces are put in a template
CredentialTemplate template = new CredentialTemplate();
template.setSecurityParameters(secParams);
template.setControlParametes(conParams):
template.setAttributeSpecification(attSpecs);

// Create the credential values object containing the actual attribute values
CredentialValues atts = new CredentialValues():
vals.add("name", "UniversalSign");
vals.add("address", "Celestijnenlaan 200A, 3000 Leuven, Belgium");
vals.validFrom(new Date()); // valid from now on

// The credential is created
Credential issuerCred = cHandler.createSelfSignedCredential(template, atts);

// The entity posessing the credential is extracted. This is possible since it
    is self signed.
// This issuer object can be published
Entity issuer = credential.getIssuer();
```

Listing 3.10: Creating a self signed X.509 credential

3.5.4 Creation of an Idemix credential template

In listing 3.11, it can be seen how a credential template can be created. In this example, the credentials will be Idemix credentials. The security parameters are made, the issuer is set, the control parameters are defined as well as the specification of the attributes that can be included in the credential corresponding to this template.

```
// Create the credential template. we start by creating the security parameters
SecurityParameters secParams = new IdemixSecurityParameters(048, 1632, 256,
    256, 1, 597, 120, 2724, 80, 160, 256, 80, 80);

// Set the issuer
Entity issuer = issuer // see previous listing;

// The control parametes are set
ControlParameters conParams = new ControlParameters();
conParams.enableAuthentication();
conParams.enableSigning();
```

```
conParams.setDuration("1 year");

// The attributes specification is set
CredentialAttributeSpecs attSpecs = new CredentialAttributeSpecs();
attSpecs.add("name", AttributeType.String);
attSpecs.add("zip", AttributeType.integer);
attSpecs.add("dob", AttributeType.integer);
attSpecs.add("gender", AttributeType.char)

// Finally, the pieces are put in a template
CredentialTemplate template = new CredentialTemplate();
template.setSecurityParameters(secParams);
template.setControlParametes(conParams):
template.setIssuer(issuer);
template.setAttributeSpecification(attSpecs);
```

Listing 3.11: Creating an Idemix credential template

3.5.5 Issue and receive a credential

In listing 3.12, it the code required by the receiver to receive a credential is shown. Evidently, this is only one of the possible scenarios. In listing 3.13, the correpsonding code to issue the credential is shown.

```
// Load credential manager and handler
CredentialManager cMgr = ....
CredentialHandler credHandler = mgr.getCredentialHandler("Idemix", "0.0.2");

// It is assumed that a connection has already been set up.
Connection conn = ...

// Retrieve the template that has been created previously
CredentialTemplate template = ...

// Create empty CredentialValues object; all values are chosen by the issuer.
CredentialValues receiverChosenAtts = new CredentialValues():

// start the protocol to receive the credential
CredReceiverTranscript rtrans = getCredential(conn, template,
    receiverChosenAtts);
Credential cred = rtrans.getCredential();
```

Listing 3.12: Receiving a credential

```
// Load credenital manager and handler
CredentialManager cMgr = ....
CredentialHandler credHandler = mgr.getCredentialHandler("Idemix", "0.0.2");
```

```
// It is assumed that a conection has already been set up.
Connection conn = ...

// retrieve the template that has been retrieved previously
CredentialTemplate template = ...

// Create the credential values object containing the attribute values
CredentialValues issuerChosenAtts = new CredentialValues():
vals.add("name", "Kristov Verslype");
vals.add("zip", 3000);
vals.add("dob", 19820101);
vals.add("gender", true);
vals.validFrom(new Date());

// execute the issue protocol.
// Complementary to the getCredential() method in the previous listing
CredIssuerTranscript itrans = issueCredential(conn, template, issuerChosenAtts)
    ;
```

Listing 3.13: Issuing a credential

3.5.6 A simple authentication protocol

Listing 3.14 and 3.15 illustrate how a prover could authenticate to a verifier. The first listing is the code executed by the prover, the second by the verifier. Afterwards, the correcness of the verification can be checked (see 3.16).

```
// Load credential manager and handler
CredentialManager cMgr = ....
CredentialHandler credHandler = mgr.getCredentialHandler("Idemix", "0.0.2");

// It is assumed that a connection handler is instantiated
// and a connection with the verifier has already been set up.
ConnectionHandler connHandler = ...
Connection conn = ...

// The credential the user will use to authenticate.
Credential cred = ...

// The template of the credential is sent to the other side
connHandler.send(conn, cred.getTemplate());

String id = cred.getTemplateDOI();
ShowSpecification spec = new ShowSpecification();
Predicate pred1 = new IntervalPredicate(id+".zip", 3000, 3010);
Predicate pred2 = new BasicPredicate(id+".dob", Relation.GREATERTHAN, 1980);
Predicate conn = new Connective(Connector.UNION);
conn.addPredicate(pred1);
```

```
conn.addPredicate(pred2);
spec.setPredicate(conn);

/**
 * The following outcommented code would only disclose
 * that the prover possesses a particular credential type.
 * If spec is empty, this is the default.
 *
 * String id = cred.getTemplateDOI();
 * ShowSpecification spec = new ShowSpecification();
 * predicate p = new PossessionPredicate(id);
 * spec.setPredicate(p);
 **/

// Send the show specification to the service provider
connHandler.send(conn, spec);

// Now, the actual authentication can be done
AuthProverTranscript trans = authenticate(conn, cred, spec);
```

Listing 3.14: Authentication

```
// Load credential manager and handler
CredentialManager cMgr = ....
CredentialHandler credHandler = mgr.getCredentialHandler("Idemix", "0.0.2");

// It is assumed that a connection handler is instantiated
// and a connection with the verififer has already been set up.
ConnectionHandler connHandler = ...
Connection conn = ...

// The service providers receives the credential template and the show
    specification
CredentialTemplate template = (CredentialTemplate)connHandler.receive(conn);
ShowSpecification spec = (ShowSpecification)connHandler.receive();

// Now, the actual authentication can be done
AuthVerifierTranscript trans = receiveAuthentication(conn, template, spec);
```

Listing 3.15: Authentication verification

```
// Load credential manager and handler
CredentialManager cMgr = ....
CredentialHandler credHandler = mgr.getCredentialHandler("Idemix", "0.0.2");

// and verify the entity which was part of the transcript
boolean result = credHandler.verifyEntity(trans.getEntity());
```

Listing 3.16: Transcript verification

3.5.7 A more complex authentication example

In the example in listing 3.17 and 3.18, the prover proves a relationship between the credential and a commitment and the credential and a verifiable encryption, and does this under a pseudonym. All relevant disclosure data is put in a `Disclosure` object, which is sent to the verifier. Since the framework prevents sending sensitive information over connections, the verifier only receives a `Disclosure` object that only allows to verify the correctness of the authentication. We emphasize that this is only one example of how prover and verifier can cooperate.

In listing 3.19, the `Authentication` object is used to do the deanonymization. Note that this object contains the `DeanonSpec` object. The resulting object contains a pseudonym or an identifier. The deanonymizer can additionally sign the result for liability/trust reasons. Listing 3.20 illustrates how the deanonymization by the TTP could be verified.

Also, a decryption of a verifiable encryption is shown in listing 3.21. Since the deanonymization techniques uses verifiable encryption, and since the denanoymizing entity knows the content of that encryption, (s)he could create a proof proving the exact value of the encryption. A decryptor of a verifiable encryption can do the same.

```
// Load credential manager and handler
CredentialManager cMgr = ....
CredentialHandler credHandler = mgr.getCredentialHandler("Idemix", "0.0.2");

// It is assumed that a connection handler is instantiated
// and a connection with the verifier has already been set up.
ConnectionHandler connHandler = ...
Connection conn = ...

// the trusted third party to do the deanonymization
Entity ttp = ...
Commitment comm = ...
VerifiableEcnryption venc = ...
Entity decryptor = ...

// The credential and the pseudonym the user will use to authenticate.
Credential cred = ...   // contains an attribute "year"
Pseudonym nym = ...   // contains an attribute "gender"

// the show specification
String idCred = cred.getTemplateDOI();
ShowSpecification spec = new ShowSpecification();
Predicate pred1 = new IntervalPredicate(idCred+".zip", 3000, 3010);
Predicate pred2 = new BasicPredicate(id+".dob", Relation.GREATERTHAN, comm.
    getName()+"year");
Predicate pred3 = new BasicPredicate(id+".gender", Relation.EQUALS, venc.
    getName()+"gender");
Predicate conn = new Connective(Connector.UNION);
conn.addPredicate(pred1);
conn.addPredicate(pred2);
conn.addPredicate(pred3);
```

```
spec.setPredicate(conn);

// The deanonymization condition
DeanonSpec deanonSpec = new DeanonSpec(cred, "name", ttp, "abuse")

ProvableDisclosure disclosure = new Disclosure();
disclosure.setShowSpecification(spec).
disclosure.addCredential(cred);
disclosure.addCommitment(comm);
disclosure.setDeanonSpec(deanonSpec);
disclosure.setPseudonym(nym);

// The template of the credential is sent to the other side
// Note that the framework takes care that no sensitive data is sent.
connHandler.send(conn, disclosure);

// Now, the actual authentication can be done
AuthProverTranscript trans = authenticate(conn, disclosure);
```

Listing 3.17: Authentication under a nym and using a commitment

```
// Load credential manager and handler
CredentialManager cMgr = ....
CredentialHandler credHandler = mgr.getCredentialHandler("Idemix", "0.0.2");

// It is assumed that a connection handler is instantiated
// and a connection with the verifier has already been set up.
ConnectionHandler connHandler = ...
Connection conn = ...

// The service providers recieves the credential template and the show
    specification
Disclosure disclosure = (Disclosure)connHandler.receive();

// Now, the actual authentication can be done
AuthVerifierTranscript trans = verifyAuthenticate(conn, disclosure);

// The verifier extracts out of the transcript the disclosed certified
    information about the prover
Entity authenticator = trans.getDisclosedEntity();
Authentication auth = (Authentication)entity.getDisclosureProof();
```

Listing 3.18: Authentication verification in which a pseudonym

```
// Load the credential handler and the connection handler
CredentialHandler credHandler = ...
ConnectionHandler connHandler = ...

// Connection is established and credential of the TTP is loaded.
```

```
Connection conn = ...
Credential ttpCred = ...

// Receive the authentication object and deanonymize it
Authentication auth = (Authentication)connHandler.receive(conn)
Deanonymization deanon = credHandler.deanonymize(auth, ttpCred);
String name = deanon.getIdentifier() // equals "Kristof Verslype"
```

Listing 3.19: Deanonymization after abuse by the trusted third party

```
Deanonymization deanon = ...
String identifier = "Kristof Verslype";
DisclosureProof auth = ...
boolean result = verifyDeanonymization(deanon, auth, identifier);
```

Listing 3.20: Verification of a deanonymization

```
// Load the credential handler and the connection handler
CredentialHandler credHandler = ...
ConnectionHandler connHandler = ...

// A connection is established and the deanonymizer key is loaded,
Connection conn = ...
VencSecretKey vsk = ...

// Receive the verifiable encryption and decrypt it
VerifiableEncryption venc = (VerifiableEncryption)connHandler.receive(conn);
boolean gender = (boolean)credHandler.decrypt(venc, vsk);
```

Listing 3.21: Decryption of a verifiable encryption received by the verifier

3.5.8 Pseudonym code examples

Receiving and issuing a pseudonym is illustrated in listings 3.22 and 3.23. Note that `nymIssuer.hasProveCapabilities()` is false, but `nymReceiver.hasProveCapabilities()` returns true. Hence, only the receiver can prove possession of the pseudonym.

In a similar way, a signed nym can be agreed. the receiver has to give an extra `Disclosure` parameter to the `receivePseudonym()` method and the issuer has to give the corresponding `Disclosure` without prove capabilities as extra parameter to the `issuePseudonym()` method.

```
CredentialManager credMgr = ...
CredentialHandler cHandler = CredentialMgr.getCredentialHandler("Idemix", "
    0.0.2");
Connection conn = ...
PseudonymTemplate = ...
NymReceiverTranscript trans = credHandler.getPseudonym(conn, template);
Pseudonym nymReceiver = trans.getPseudonym();
```

Listing 3.22: Receiving a pseudonym

```
CredentialManager credMgr = ...
CredentialHandler cHandler = CredentialMgr.getCredentialHandler("Idemix", "
    0.0.2");
Connection conn = ...
PseudonymTemplate = ...
NymIssuerTranscript trans = credHandler.issuePseudonym(conn, template);
Pseudonym nymIssuer = trans.getPseudonym();
```

Listing 3.23: Issuing a pseudonym

Note that the the pseudonym template will be provided by the service provider and can be determined based on values found in a credential template. In that case, the provider developer can offer a `IdemixPseudonymTemplate` class in which an Idemix credential template is given as parameter to the constructor. Separate `PseudonymParameter` objects could be generated as well.

3.6 Other Components

3.6.1 Privacy Handler & Manager

The task of the privacy manager is to keep track of the personal information that has been disclosed to other entities. On the other hand, it could keep track of the personal information the service provider knows about other entities. Therefore, it is necessary that the framework submits each transcript to the privacy manager. Based on these transcripts, the user or service provider can be informed about the disclosed personal information. Information could be disclosed under an identifier or a pseudonym, allowing to link different transcripts to the same entity. Depending on the type of the underlying connection, disclosed data could be linked to each other if they are disclosed over the same connection.

Secondly, the privacy manager could measure the user's privacy under certain pseudonyms or if certain properties are disclosed or linkable. This indeed influences the anonymity set. The privacy manager could provide techniques or heuristics to measure these anonymity sets and inform the user about it. This level of anonymity can be towards a single party or towards a set of (potentially colluding) parties. Also deanonymizers can be part of such a set.

Depending on the policy, actions can be taken automatically, actions can be forbidden, or user consent might be requested.

The privacy manager policy could define masks towards certain parties or set of parties, meaning that some parties are more trusted and allowed to receive certain personal information, while the same data is hidden or made more coarse grained towards others. For instance some health data is only disclosed to trusted parties belonging to the medical domain.

The privacy manager could also keep track of context or application specific data in locally managed profiles. These profiles can be queried by external parties. The type of queries and the fine-grainedness of the results is defined in the policy related to the privacy mananger. For instance, a user could keep track of the books he bought and the books he is interested in. An eHealth bookstore may ony be allowed to see the medical books bought by the user or in what medical books the user is interested in. A general bookstore does not have access to this information and a general eCommerce site may only see the book categories in which the user is interested.

The privacy handler offers the techniques to measure or estimate the user's privacy. This could be based on publicly available statistical information for disclosed properties, but also the anonymity provided by the underlying network should be taken into account. Secondly,it could provide an implementation for the local profiles.

3.6.2 Dispute Handler & Manager

If people are anonymous, this anonymity can often be abused. Therefore, anonymity can be conditional, meaning that a user can be deanonymized. The ticketing application in appendix A illustrates that multiple parties need to be involved in order to deanonymize someone after misbehaving.

The dispute manager of the different parties should be configured such that evidence is stored in case of complaints and that on the other hand evidence is stored to be able to prove that other parties indeed made errors. As another example, anonymous prescriptions could be given.

The dispute manager should also know the logic of doing a deanonymization and inform the user about that process. The user can be given the choice between multiple deanonymization paths, depending on the trust he puts in the different links in the deanonymization chains.

The dispute manager should also provide the possibility that other parties query the user's evidence. However, only a very limited number of highly trusted parties are allowed to do this. This way, the user is not bothered if a dispute needs to be solved. This is again a policy issue.

3.7 Using Framework on Mobile Devices

The framework helps users to maintain their credentials and gives them control over their privacy. The current implementation of the framework doesn't allow the user to run it on a mobile device (e.g. smartphone). However, this can be useful because users don't always have a laptop or desktop in their neighbourhood if they use credentials (e.g. a loyalty card in a supermarket, train ticket, badge to enter parking lot).

Mobile devices have less computational power and don't support Java or .NET programs but only offer a limited interface to the programmers of mobile applications (e.g. Java Mobile Edition and .NET Framework Compact Edition). Hence, the framework needs to compile on Java ME to be able to run on mobile devices. First, the framework core needs to be rewritten so that it only uses the subset of Java classes that are available in Java ME. Second, new providers need to be implemented that use the cryptographic methods that are defined in Java ME. This can be a very tough job (e.g. the class BigInteger is not available in Java ME while Idemix requires such a class). However, all applications that use the framework to use credential don't need any changes in the source code to run on the mobile devices. This only holds for the source code where the framework is used. Probably, the user interface of the application needs some adaption to run on a mobile device.

As mobile devices are equipped with more types of communication technologies (e.g. Bluetooth, WiFi, 3G, NFC), the framework needs to take care of the security and privacy properties between the different technologies. Moreover, a user may want to specify that he only wants to use some credentials over a short-distance communication (like NFC) or that 3G may only be used if WiFi is not available. This needs a policy language that lets the

user specify his preferences and requirements and a good policy manager in the framework that can make a decision (allowing, proposing or denying certain communication channels depending on which credential is used).

Chapter 4

Validation

4.1 Validation based on eTicketing

In this section, an eTicketing application is build using the framework. The interface in the framework should make the implementation easier (because it makes abstraction of the used technology). The protocol of the eTicketing application is defined in appendix A.

4.1.1 High-Level Description

Before starting the implementation, an overview of the eTicketing application must be made with the actors in the system.

Actors

The ticketing application exists of 3 actors: the *buyer*, the *seller* of tickets (ticket shop) and the *issuer* of anonymous credentials. The government issues Belgian eID cards. However, this phase is not within the scope of this example.

The sample application consists of 2 phases:

1. The buyer authenticates with his eID to the issuer. If authentication is successful and if the buyer didn't receive an anonymous credential yet, the issuer issues an anonymous credential to the buyer.

2. The buyer requests an event list to the ticket shop. To buy tickets, he sends a request to the ticket shop with the eventID and the number of tickets he wants to buy. If tickets are still available, the ticket shop sends a list of ticket numbers to the buyer. The buyer uses his anonymous credential to sign the ticket numbers. The ticket shop checks the signature and the validity of the anonymous credential. The payment should be realized in this phase. However, this is not implemented in this example.

4.1.2 Usage of the framework

The application consists of three modules (i.e. one module per actor). First, common concepts are introduced (like a credential template, a transcript and an identity file). Next, the

implementation of the *issuer* and the *ticket shop* are discussed. Finally, the *client module* is presented.

Issuer module

The issuer listens for incoming connections, authenticates users and issues anonymous credentials. The source code below lists the source code of the issuer. Note that the framework is instantiated first (by creating a new `Framework` object that reads out the identity file).

```
Framework framework = new Framework(new File("issuer.idf"));
```

The issuer will keep a list of users that have already registered.

```
ArrayList<String> registeredUsers = new ArrayList<String>();
```

The issuer uses the `CommunicationHandler` to listen for incoming connections on TCP port 2001.

```
CommunicationSettings commSettings =
  new SocketCommunicationSettings("0.0.0.0", 2001);
CommunicationManager commManager = framework.getCommunicationManager();
CommunicationHandler commHandler =
  commManager.getCommunicationHandler(commSettings.getTechnology());
ConnectionListener listener = commHandler.startListen(commSettings);
while (true){
  Connection conn = commHandler.accept(listener);
```

If a client connects to the issuer, the issuer will require client authentication with the Belgian eID card. Therefore, he instantiates the corresponding `CredentialHandler` and calls the `receiveAuthentication` method. If the authentication fails, an exception is thrown. So, if the `receiveAuthentication` method returns a transcript object, the authentication was successful.

```
  Technology beid = new Technology("BelgianEid", new Version(0,0,1));
  ch = framework.getCredentialManager().getCredentialHandler(beid);
  AuthVerifierTranscript avt = ch.receiveAuthentication(conn, null);
```

At authentication, the issuer reads out the certain identifying data.

```
  String uniqueID = (String) avt.getReleasedAttributes().getValue("uniqueID");
  int age = (Integer) avt.getReleasedAttributes().getValue("age");
```

The issuer checks if that user already received an anonymous credential.

```
  if(registeredUsers.contains(nrn)) {
    commHandler.sendObject("Already registered");
  } else {
    commHandler.sendObject("OK");
```

Before the issuer can issue Idemix credentials, he has to load his own credential using the `StorageManager`.

```
    StorageManager sm = identity.getStorageManager();
    Credential issuerCred = sm.loadCredential(1);
```

An Idemix credential is issued by using the corresponding `CredentialHandler`. A credential template is loaded from XML. The attribute values are assigned.

```
    ch = framework.getCredentialManager().getCredentialHandler(idemix);
    Template template = new Template(new File("templateCredential.xml"));
    values = new AttributeValues();
    values.add(new AttributeValue("age", age));
    IssueTranscript it = ch.issueCredential(conn,template,values,issuerCred);
```

The user is added to the list of registered users in order to prevent that he can ask a another Idemix credential. At the end of the program, the issuer closes the connection.

```
    registeredUsers.add(uniqueID);
  }
  commHandler.disconnect(conn);
}
```

Ticket shop module

The ticket shop module gives an overview of available tickets for all the events and handles purchases. The identity file of the ticket shop is first loaded and the server is listening for incoming connections (in this example on TCP port 2002).

```
Framework framework = new Framework(new File("ticketshop.idf"));
CommunicationSettings commSettings =
  new SocketCommunicationSettings("0.0.0.0", 2002);
CommunicationManager commManager = framework.getCommunicationManager();
CommunicationHandler commHandler =
  commManager.getCommunicationHandler(commSettings.getTechnology());
ConnectionListener listener = commHandler.startListen(commSettings);
while (true){
  Connection conn = commHandler.accept(listener);
```

The ticket shop module can receive two types of messages from the user, namely (1) a request for seat that are available for a specific event or (2) a request to buy some tickets. The seat list is included in an `Event` object in this example. The implementation of the `Event` object is not relevant for this example application.

```
  String message = (String) commHandler.receiveObject(conn);
  if(message.equals("freeseats")){
    ArrayList<Event> events = loadEventList();
    commHandler.sendObject(events);
  } else {
```

If the user wants to buy some tickets, he needs to sign the ticket numbers or seat numbers with his Idemix credential while proving that he is older than 18.

```
    Technology idemix = new Technology("Idemix", new Version(0,0,2));
    ch = framework.getCredentialManager().getCredentialHandler(idemix);
    Template userCredentialTemplate = new Template(new File("template.xml"));
    ShowSpecification spec = new ShowSpecification();
    spec.setAssertions(new BasicPredicate(userCredentialTemplate.getName(),
      "age", Operator.greaterOrEqual, 18));
    String message = (String) commHandler.receiveObject(conn);
    // check if message is valid and seats are still available
    SignVerifierTranscript svt =
      ch.verifySignature(conn, userCredentialTemplate, spec, message);
  }
  commHandler.disconnect(conn);
}
```

Buyer module

A user first needs to register with his eID card and request an anonymous credential. With that credential, he must be able to buy tickets at the ticket shop. The first step in the implementation is loading the identity file and connecting to the issuer host.

```
Identity identity = new Identity(new File("buyer.idf"));
CommunicationSettings commSettings =
  new SocketCommunicationSettings("issuerHost", 2001);
CommunicationManager commManager = identity.getCommunicationManager();
CommunicationHandler commHandler =
  commManager.getCommunicationHandler(commSettings.getTechnology());
Connection conn = commHandler.connect(commSettings);
```

After the connection is established, the Belgian eID is used to authenticate to the issuer.

```
Technology beid = new Technology("BelgianEid", new Version(0,0,1));
ch = framework.getCredentialManager().getCredentialHandler(beid);
Credential cred = BelgianEidCredentialHandler.getCredentialObject();
AuthProverTranscript apt = ch.authenticate(conn, cred);
```

Thereafter, an Idemix credential is issued to the buyer.

```
Technology idemix = new Technology("Idemix", new Version(0,0,2));
ch = framework.getCredentialManager().getCredentialHandler(idemix);
Template template = new Template(new File("templateCredential.xml"));
GetCredentialResult result = ch.getCredential(conn, template, null);
Credential credential = result.getCredential();
// user should store his credential
```

The buyer closes the connection with the issuer and connects to the ticket shop.

```
commHandler.disconnect(conn);
commSettings = new SocketCommunicationSettings("ticketShopHost", 2002);
conn = commHandler.connect(commSettings);
```

The user can request the available seats for all events and select certain tickets. The user signs his ticket and/or seat numbers with his Idemix credential. It is important for Idemix that the `ShowSpecification` is the same at both the client and server side.

```
commHandler.sendObject("freeseats");
ArrayList<Event> events = (ArrayList<Event>) commHandler.receiveObject(conn);
// show list of free seats to user
String message = "Concert: 12-2, 12-4, 12-6" // string of selected seats
ShowSpecification spec = new ShowSpecification();
spec.setAssertions(new BasicPredicate(userCredentialTemplate.getName(),
  "age", Operator.greaterOrEqual, 18));
SignProverTranscript spt = ch.sign(conn, credential, spec, message);
commHandler.disconnect(conn);
```

4.1.3 Evaluation

Improvements by the Framework

By creating a configuration file and a framework instance for each actor, the developer is encouraged to define a list of actors for his application. The high-level methods in the framework (e.g. authenticate, sign) keeps the developer away from the security protocols that are used. He doesn't need to handle revocation status, challenges, trust chains, etc. Moreover, if the underlying security protocols need an update of patch, the source code of the application can remain, only an updated or new provider that is plugged into the framework is necessary.

Framework Manager Support

Currently, this example application only makes use of the managers in the framework to get access to the correct handler that implements a specific technology. When more functionality is implemented in the managers, they could help the user in selecting the correct credential. In this example, the seller could allow a set of credentials (e.g. drivers license, identity card) and the credential manager in the framework can then choose a valid credential in the repository of the buyer.

Framework Handler Improvements

During the implementation of this example application, no `Request` or `Response` objects were available in the framework. They have now been included in the framework to ease the construction of messages between client and server and to handle error messages more properly.

4.2 Validation based on ePoll

In this section, the framework is validated with some more advanced application: ePoll. The ePoll application allows a user to fill in a poll anonymously. However, the same user can only fill in the same poll once. For some polls, the user must proof that he fulfill some requirements (e.g. older than 18 years, live in Leuven, female). Optionally, the user can reveal some more

information if he wants. This way, the poll organiser can create some more advanced statistics. More information about the ePoll protocol is available in appendix B.

4.2.1 High-Level Description

High level Description

High level description of the application

Roles and Interactions

Different roles and their functionality/interactions. Especially w.r.t to the framework functionality.

4.2.2 Usage of the framework

Data objectes in the system (keys, credentials, transcripts, commitments, psuedonyms, etc) and how they are created, stored, used and managed by the framework. Interesting code sequences of the application using the API can be included. Proposal:

I would suggest one subsubsection per relevant data object. in the ePoll case for instance; one for the user's sign credential, one for the transcript stored by the verifier, etc. Other suggestions are welcome. Example code of how the application uses the framework can however involve multiple data objects.

4.2.3 Evaluation

Improvements by the Framwork

Discuss improvements over that application if it were developed without using the framework. For instance, about what things didn't the developer need to bother. How did the framework help in the development. Alo stress the flexibility of the framework.

Efficiency (OPTIONAL)

optionally, test results (measurements) can be included.

Framework Manager Support

what could the managers in the framework have done to further facilitate/enhance the development of the application?

Framework Handler Improvements

Where could the framework handler be improved based on the experience using this application

Chapter 5

Modelling and Synthesizing Privacy-Preserving Applications

5.1 Premise

The proposed framework strives to and succeeds in providing "a uniform interface to facilitate the development of privacy preserving applications". It enables a programmer with extensive to basic understanding of cryptographic- and security-related concepts to implement privacy preserving applications. The framework can thus – simplistically – be viewed as a means to enable programmers to do more with less hassle. From here arises an interesting question – especially under the premise that cryptography and/or security experts are harder to come by than able programmers – : what can be done to enable domain experts to do more with less hassle? More specifically, what can be done to enable cryptography and/or security experts with little to no programming capabilities to develop privacy preserving applications?

Domain-Specific Modelling (DSM) is a relatively young discipline whereby applications[1] are modelled at a high-level of abstraction using constructs that are tightly coupled to some restricted domain's concepts [25, 19, 6]. Subsequent *Model Transformations* transform the models into a collection of low-level artifacts that form the final applications. The advantages of using model-driven approaches as opposed to code-centric approaches to application development are numerous; some of them are listed below.

- Domain experts can model applications/protocols/etc. using notions familiar to them. This implies that interaction with a programmer and/or concept translation to code-related constructs (e.g. classes and functions) are longer necessary.
- Code-generation from models makes it possible to quickly develop robust applications and makes the fact that small conceptual changes (at the model level) may cause significant changes in the code irrelevant, thereby speeding up not only development but also maintenance and evolution. Already, several real-world experiments using DSM for

[1] A wide variety of artifacts can be modelled ranging from configuration files, to data schemas, to protocols, to partial or complete applications. For brevity, we restrict our discussion of DSM to the modelling and synthesis of applications.

small to medium scale development efforts have reported decreases of up to an order of magnitude in their development times [8, 32, 25].

- Appropriately designed model transformations enable targeting code-generation to different platforms (e.g. PDAs, Internet Browsers, etc.) without any changes to the models.
- Models are often easier to simulate and analyze (e.g. possibility of deadlocks, possibility of anonymity break due to usage patterns) then coded applications.

Despite the many advantages of DSM, it is by no means a magical technique that enables the synthesis of any functionality from a few mouse clicks. For instance, although we argue that privacy preserving applications should be modelled by domain experts and synthesized automatically rather than coded manually by programmers, the necessity of some sort of library to support the modelled security-related operations remains. For instance, a model of a privacy preserving application which depicts the anonymous authentication of two parties and subsequent secure transfer of data requires that authentication and transfer services be available. Thus the relationship between the proposed framework and our modelling techniques is revealed: the synthesized applications will contain appropriate code snippets to make use of the framework.

5.2 Case Study: Prescription Issuing Protocol

Due to the abstract nature of DSM, we begin by introducing an example which we will use to clarify and strengthen some of our previous allegations. Our example describes the issuing a prescription from a doctor to a patient. We will present it in three different forms: in layman's terms, as understood by a cryptography expert and as it could be modelled in a DSM tool.

The prescription issuing protocol can be described quite simply by a few interactions between patient and doctor[2]:

1. The patient P electronically contacts someone he believes to be doctor D;
2. D authenticates himself as a certified physician without revealing his identity;
3. P authenticates himself as a valid patient (e.g. insured) without revealing his identity;
4. Using P's credentials, D verifies that P should receive the requested medicine M;
5. D writes and securely transmits a prescription to P such that it may only be used n times and only by P;
6. P takes whatever steps are necessary to receive M (e.g. communicate with a pharmacist, etc.).

[2]For brevity, we leave out the subsequent interaction between patient and pharmacist.

Figure 5.1 shows how this protocol is described by a cryptography expert. The bidirectional arrow between `Dr.` and `Patient` represents steps 1 through 4. The following arrow from `Dr.` to `Patient` represents step 5. Finally, the rest of the diagram represents step 6 (the interaction between patient and pharmacist).

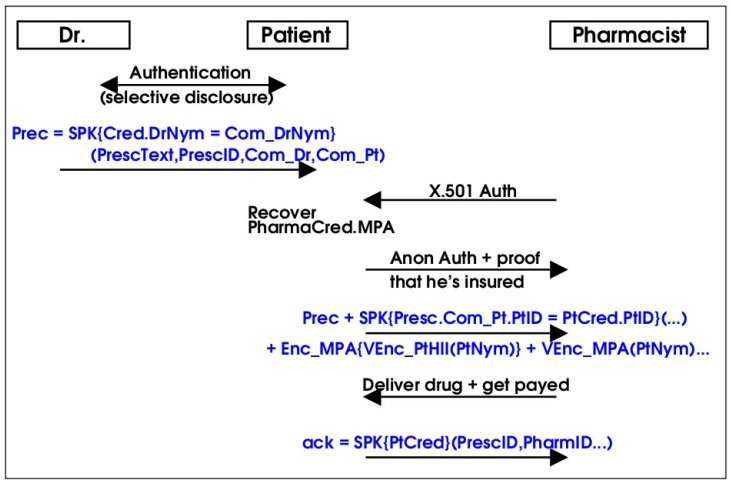

Figure 5.1: Prescription Issuing described by a cryptography expert

Let us now turn our attention towards Figure 5.2 which shows how the protocol can be modelled in a DSM tool tailored for cryptographic protocol and privacy preserving application designers. The concepts of *parties* and their *actions* are represented by two instances of the `ActorApplication` construct, `DoctorApplication` and `PatientApplication`, and their associated `LifeLine` instances respectively. The concept of *selective disclosure* is embodied by instances of the `ReadData` construct which enables the specification of arbitrary data sources (e.g. user input or Belgian eID card) and attributes to be read. The concept of *authentication* is comprised within instances of the `ShowCredential` construct which can be customized to use arbitrary data sources, credential types and multiplicities, and certification authorities. The concept of secure transfer of data is encapsulated within instances of the `SendData` construct. Finally, instances of the `InformationMessage` construct enable the modeller to include arbitrary text in the application. In our example model, these are used for "introductory" and "concluding" messages.

The crucial element we wish to point out by introducing these various representations of the prescription issuing protocol is the strong similarity between the modelled representation of the protocol and the cryptographer's mental model. A one to one relationship between the constructs in both representations can be readily established which we argue shows that the proposed modelling language is indeed at the level of abstraction of how cryptography experts choose to describe their protocols. Furthermore, the constructs in the modelling language are sufficiently customizable to enable the generation of a wide variety of privacy preserving applications and to properly populate the required function calls to the underlying framework.

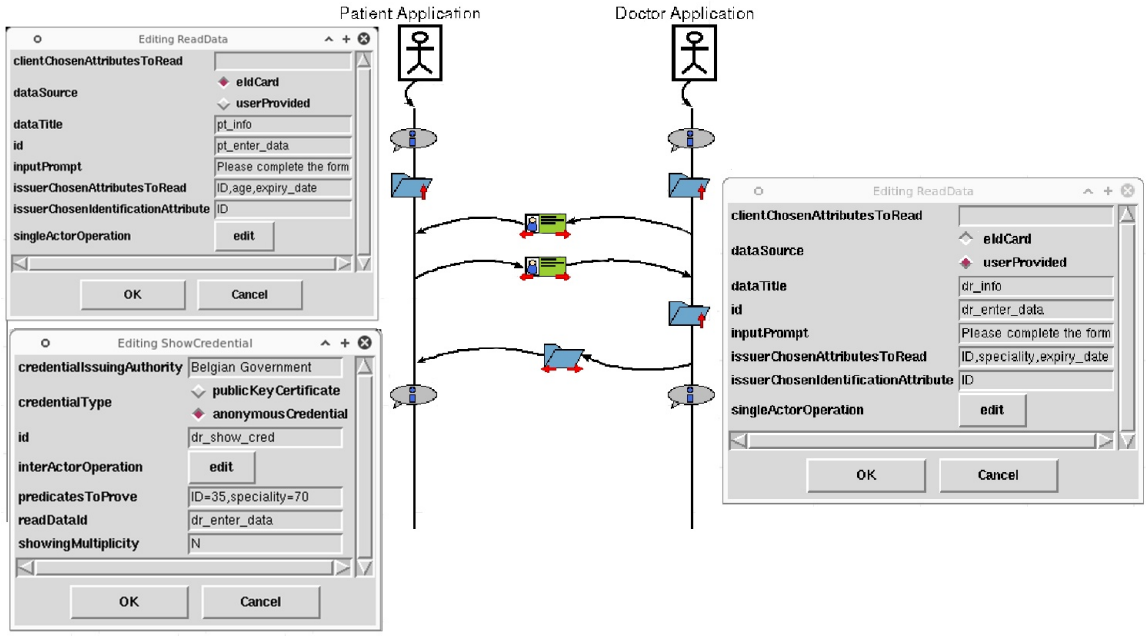

Figure 5.2: Prescription Issuing modelled in a DSM tool. The stick figures are instances of the `ActorApplication` construct; the vertical black lines are instances of the `LifeLine` construct; the information bubbles are instances of the `InformationMessage` construct; the blue folders with upward arrows are instances of the `ReadData` construct; the ID cards with sideways arrows are instances of the `ShowCredential` construct; and the blue folders with sideways arrows are instances of the `SendData` construct.

5.3 Synthesizing Applications from Models

We targeted two distinct platforms for our generated applications: Java applets (that can be embedded and run in web browsers) and Google Android (an operating system designed for mobile devices). As mentioned earlier when listing the benefits of DSM, a single high-level model is used to generate applications for various platforms (i.e. the models need not be altered based on the target).

5.3.1 The Domain-Specific Modelling Language

Before detailing the steps that bring us from a high-level model of a privacy preserving application to the application itself, we will formally describe the modelling language we propose. Figure 5.3 shows the UML class diagram that defines our modelling language. Table C.1 details the purpose, meaning and syntax of its classes and relationships. Finally, Table C.2 details the purpose, meaning and syntax of the classes' attributes. Note that in the interest of brevity, we omitted an important part of any modelling language: syntactic

and semantic constraints. For instance, though these do not appear in our class diagram, we define several constraints to ensure such things as uniqueness of `id` attributes and "temporal consistency" (e.g. a `ReadData` instance shouldn't be referenced before it appears).

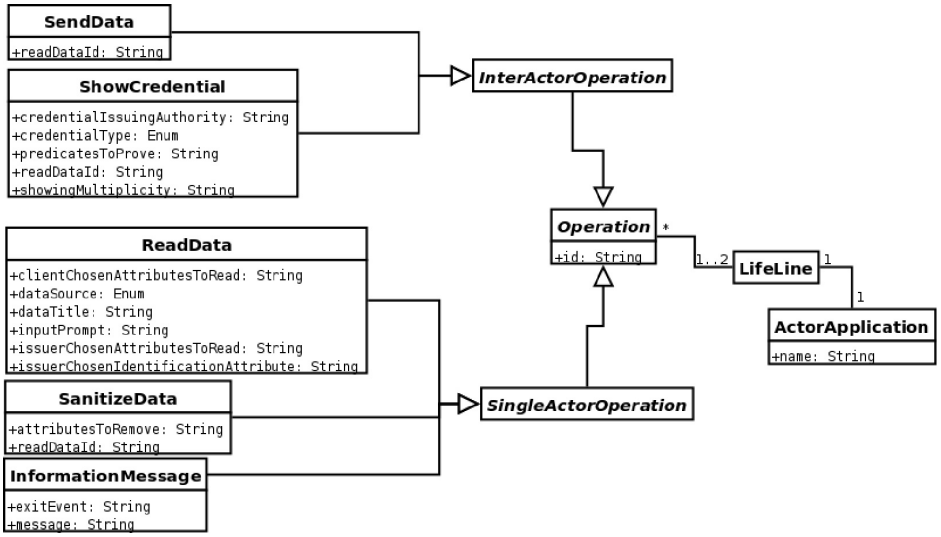

Figure 5.3: The UML class diagram that defines the proposed privacy preserving application modelling language.

5.3.2 Model Transformations

There are a wide variety of means of synthesizing low-level artifacts from models [15]. Our approach consists in dividing the transformation process into modular rules which contain a *left-hand side* (what we wish to match), a *right-hand side* (what we wish to produce in place of what we matched), an *application condition* (some snippet of code which must complete successfully for the rule to be applicable) and a *priority* (which is used to arbitrarily order the execution of rules). Due to the flagrant similarities between the transformation rules that synthesize Java applets and Google Android applications from our domain-specific models, we will restrict ourselves to a discussion on how to synthesize Google Android applications. First, we introduce `PhoneApps`, a *meta-model* – simplistically put, a modelling language definition – that we recently proposed for modelling mobile device applications [26].

The `PhoneApps` Meta-Model

Mobile device applications often require high levels of user interaction. It can thus be argued that behavior and visual structure make up the domain of such applications. The `PhoneApps` meta-model encompasses both of these aspects at an appropriate level of abstraction. Figure 5.4 shows its class diagram. Essentially, timed, logical and user prompted transitions describe the flow of control between `Containers` – that can contain other `Containers` and `Widgets` – and `Actions` – mobile device specific features – with each screen in the final application modelled as a `Container` contained in no other. With a series of graphically defined model transformations rules, `PhoneApps` models are translated to increasingly lower

level formalisms until a complete Google Android application is synthesized. Figure 5.5 gives an overview of the hierarchical relationships between the meta-models in play. For more information on artifact synthesis from `PhoneApps` models, see [26].

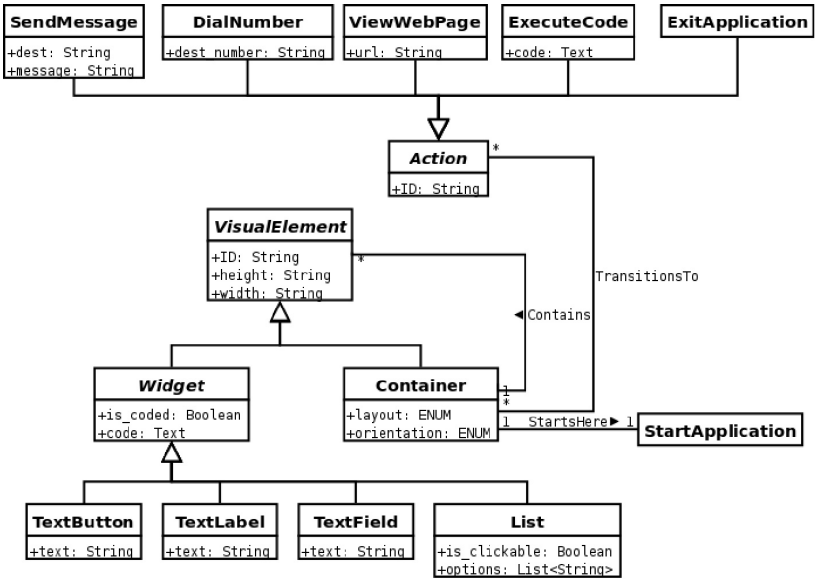

Figure 5.4: The UML class diagram that defines the proposed mobile device application modelling language.

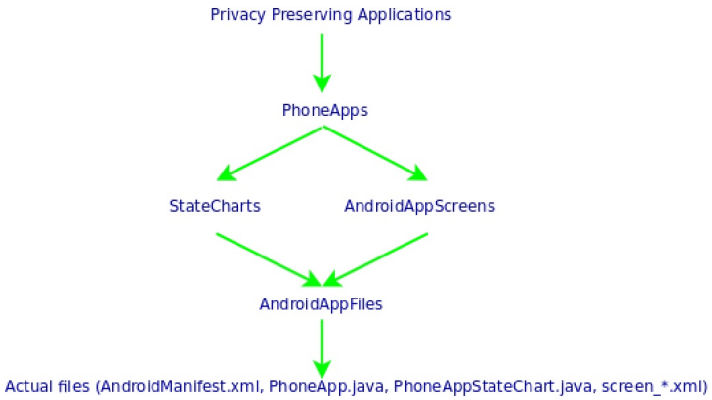

Figure 5.5: The trace map of the formalisms in play with arrows indicating transformations from one to the other.

From Privacy Preserving Application Models to `PhoneApps` Models

The process of transforming our domain-specific models into `PhoneApps` models is comprised of 12 model transformation rules which are listed and described in Table 5.1. Note that although many rules have priority 3, only one of them is applicable at any given time. This is because each rule is equipped with an application condition which restricts pattern matching

to the top-most unhandled `Operation`. For example, if the top-most `Operation` that hadn't yet been handled by a priority 3 rule on the chosen party's `LifeLine` were to be an instance of the `SanitizeData` construct, only the `SanitizeData2PhoneApps` rule would be applicable and would thus be chosen to run after which some other rule would become applicable and run and so on and so forth. See Figures 5.6 and 5.7 for concrete illustrations of two of the described rules, `DelayInformationMessage2PhoneApps` and `ReadDataFromEidCard2PhoneApps`.

Rule	Priority	Description
`ChooseApplicationToGenerate`	1	Chooses a random or arbitrary party to generate the application for.
`FirstOperation2PhoneApps`	2	Initializes the `PhoneApps` model.
`ClickInformationMessage2PhoneApps`	3	Generates appropriate `PhoneApps` constructs for an instance of the `InformationMessage` construct where the `exitEvent` attribute is set to *userClick*.
`DelayInformationMessage2PhoneApps`	3	Generates appropriate `PhoneApps` constructs for an instance of the `InformationMessage` construct where the `exitEvent` attribute is a numeric delay.
`ReadDataFromEidCard2PhoneApps`	3	Generates appropriate `PhoneApps` constructs for an instance of the `ReadData` construct where the `dataSource` attribute is set to *eIDCard*.
`ReadDataFromUser2PhoneApps`	3	Generates appropriate `PhoneApps` constructs for an instance of the `ReadData` construct where the `dataSource` attribute is set to *userProvided*.
`SanitizeData2PhoneApps`	3	Generates appropriate `PhoneApps` constructs for an instance of the `SanitizeData` construct.
`SendData2PhoneApps_Src`	3	Generates appropriate `PhoneApps` constructs (from the data sender's perspective) for an instance of the `SendData` construct.
`SendData2PhoneApps_Dest`	3	Generates appropriate `PhoneApps` constructs (from the data receiver's perspective) for an instance of the `SendData` construct.
`ShowCredential2PhoneApps_Src`	3	Generates appropriate `PhoneApps` constructs (from the credential "shower"'s perspective) for an instance of the `ShowCredential` construct.
`ShowCredential2PhoneApps_Dest`	3	Generates appropriate `PhoneApps` constructs (from the credential verifier's perspective) for an instance of the `ShowCredential` construct.
`LastOperation2PhoneApps`	4	Finalizes the `PhoneApps` model.

Table 5.1: The 12 rules that form the complete transformation from our privacy preserving application models to `PhoneApps` models.

5.4 Evaluation

In this section, we evaluate our approach. We begin by presenting photographs of our synthesized applications and move on to a short discussion about the experienced versus

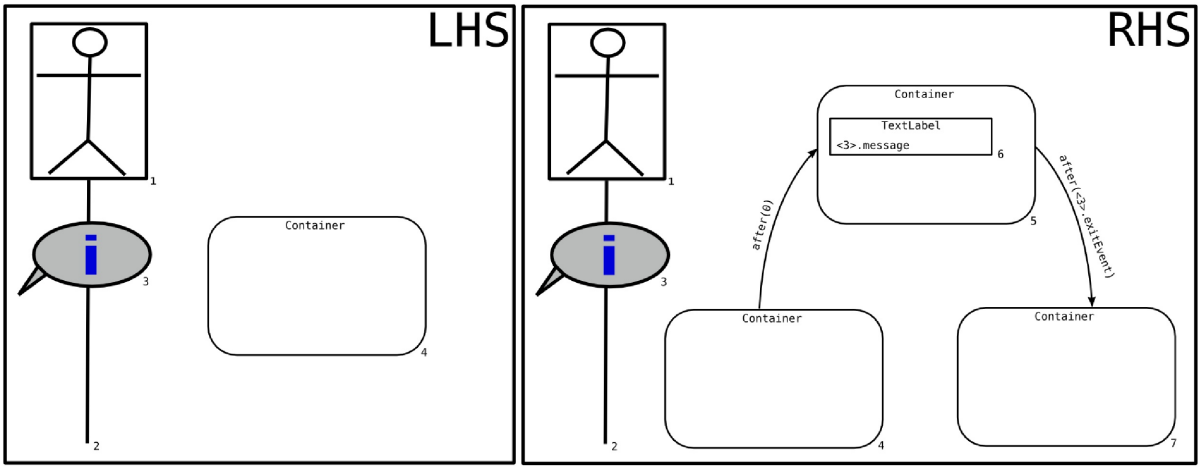

Figure 5.6: The model transformation rule `DelayInformationMessage2PhoneApps` as seen in our DSM tool. A `PhoneApps` `Container` containing a `TextLabel` with the text specified by the `message` attribute of the matched `InformationMessage` is created. It transitions to a *dummy* `Container` after the delay specified within the `exitEvent` attribute of the matched `InformationMessage`. Note that both `Containers` 4 and 7 are *dummy* `Containers` which we use to facilitate the connection of the first `Container` of one rule to the last `Container` of the previous rule.

advertised benefits of using DSM.

5.4.1 Synthesized Applications

Figures 5.8 and 5.9 show samples of the generated applications running on a real HTC Magic phone, a Google Android enabled device. In Figure 5.8, we show a "welcome screen" – which corresponds to the `InformationMessage` instance at the top of the patient's `LifeLine` in Figure 5.2 –, and two eID card related screens – which correspond to the `ReadData` instance of the patient's `LifeLine` in Figure 5.2 –. The two latter screens were generated by the `ReadDataFromEidCard2PhoneApps` model transformation rule, hence the similarities between them and the right-hand side `Containers` from Figure 5.7. In Figure 5.9, we show a "welcome screen" – which corresponds to the `InformationMessage` instance at the top of the doctor's `LifeLine` in Figure 5.2 –, and a data entry screen – which correspond to the `ReadData` instance of the doctor's `LifeLine` in Figure 5.2 –. Finally, notice how a minute change at the domain-specific model level (i.e. changing the `dataSource` attribute from *eIDCard* to *userProvided*) can result in significantly different generated applications.

5.4.2 Benefits of DSM

The development time speed-ups and raise of abstraction levels promised by DSM were indeed confirmed. A cryptography protocol expert and co-author of this document, Mohamed Layouni, required only a few minutes to familiarize himself with the domain-specific language

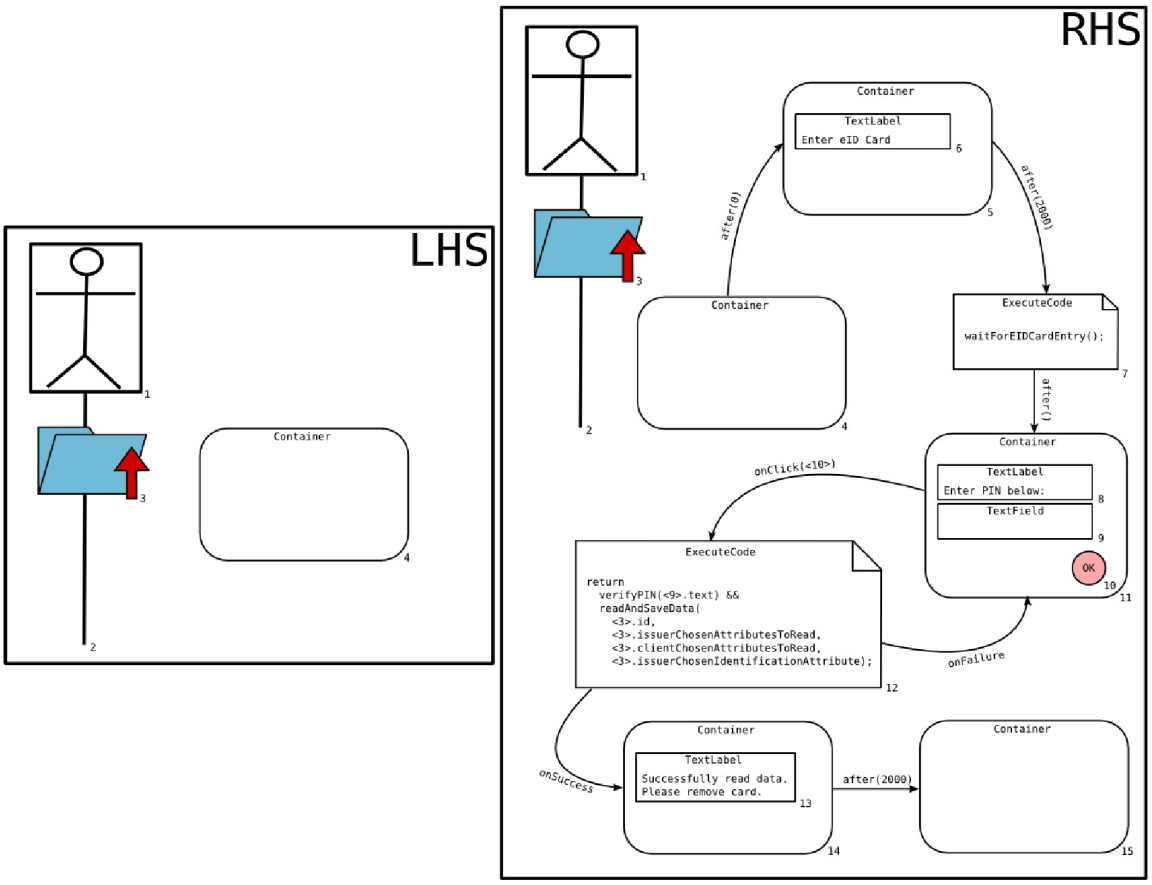

Figure 5.7: The model transformation rule `ReadDataFromEidCard2PhoneApps` as seen in our DSM tool. Three `PhoneApps` `Containers` containing instructions pertaining to the interaction with a Belgian eID card are created. Between them, two `ExecuteCode` constructs are inserted; the first one waits for the insertion of a valid Belgian eID card before proceeding while the second verifies the user provided PIN against the one stored on the card and reads the data specified in the matched `ReadData` from the card. Note that `ExecuteCode` instances containing calls to the proposed framework are generated by other rules, most notably `ShowCredential2PhoneApps_Src` and `ShowCredential2PhoneApps_Dest`.

due to its strong coupling with notions from his own domain. Furthermore, he was able to design and create the model from Figure 5.2 within less than ten minutes whereas programming two separate bug free applications – one for the patient and one for the doctor – on two different platforms – Java applets and Google Android – would have taken an intermediate programmer such as himself several hours if not days.

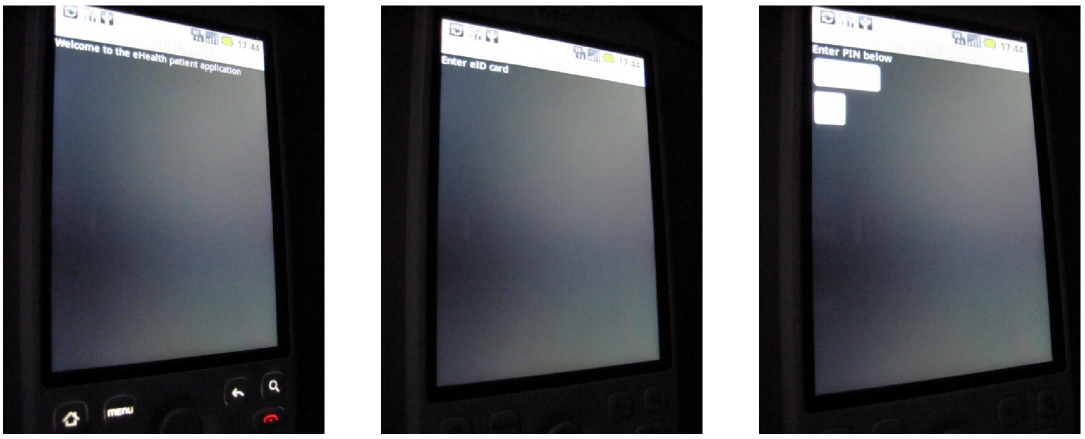

Figure 5.8: Generated patient application running on an *HTC Magic* phone

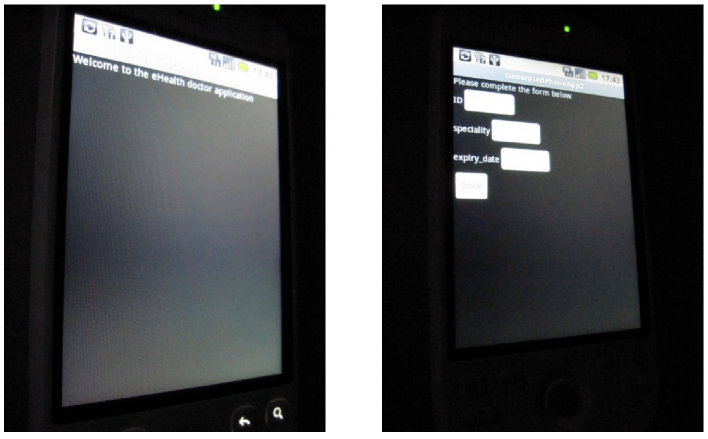

Figure 5.9: Generated doctor application running on an *HTC Magic* phone

Chapter 6

Conclusions and Future Work

6.1 The Adapid Framework

In this deliverable, a framework to assist the application developer to integrate privacy in his client-server applications has been presented. The focus is on the disclosure of personal certified properties of the user over the network. Therefore, a uniform interface has been proposed to establish connections, to issue and use credentials and to store credentials and other related data. Providers have been made to implement these interfaces. For instance support for the Belgian eID card, X.509 certififcates and Idemix are present. Although not all the technologies that can be plugged into the framework must be privacy enhancing, it is very easy to do so. The application developer only has to do minimal changes in order to mke his application more provacy enhancing if better implmentations for the framework are released.. The focus of this deliverable was on the so called handlers, the interfaces for a family of technologies (connections, persistence and credentials). The validation has shown that the framework indeed offers a significant gain to the application developer.

However, still, a lot of future work is possible. The privacy manager and handler will be considered, since they are of key importance in order to keep the user informed about his anonymity towards other parties. The handlers offer lower level interfaces, but on top of these handlers, higher level functionality can be offered as sketched in the deliverable. Another aspect for future work is running the framework on mobile devices. Therefore, performance is of key importance. As mentioned several times in the deliverable, policies need to be developed for the framework. Based on the input of application developers, a new iteration can be made. For instance export and import functionality is still only to a limited extend present in the current version. With the current framework, some steps have been made, but this paragraph tries to convince the reader that there are still a lot of open issues.

6.2 Modelling and Synthesizing

Domain-specific modelling has proven itself in the past to be a good means of leveraging the different expertise of programmers and non-programmers as well as considerably reducing development times. The large conceptual gap between the domains of privacy-preserving eServices and reactive applications as well the commonalities between various eServices make them prime candidates for the application of DSM techniques. We have proposed and demon-

strated a means of elevating the accessibility of the proposed framework to non-programmer cryptography experts by means of a tailored DSM language and environment.

A few key elements need to be developed further to fully benefit from the use of DSM in this context. First, extending the primitives available in the modeling environment will enable a wider variety of privacy preserving applications to be modeled. Second, extending the model transformation rules will enable a wider variety of platforms to be targeted and for the generated applications to be more polished. Finally, model analyses and simulation facilities should be developed to assert certain protocol properties such as linkability.

Appendices

Appendix A

eTicketing

A.1 Introduction

Tickets are used for an innumerable number of events: soccer matches, music festivals, exhibitions, etc. These tickets are ever more bought electronically. An increasing number of countries issue electronic identity cards to their citizens. Examples are Belgium, Estonia and Austria. These eID cards usually allow the holder to authenticate and to digitally sign documents, but often, they are very privacy unfriendly. For example, authentication using the Belgian eID card will usually lead to the divulgement of important personal data such as your national registration number (NRN). Despite these privacy dangers, the use of the eID card is promoted by the governments. We can thus expect that in the near future, electronic ticketing systems will arise based on the eID card. A trivial solution is easy to devise. However, this solution is not acceptable because it further endangers the card holder's privacy as profiles can easily be compiled, linked to each other and to the identity of the card holder. An advantage of the use of eID cards is that it is straightforward to impose restrictions on the maximum number of tickets that can be bought by one user, hence, thwarting the sales on black markets. Sometimes, special offers are available for buyers under or over a certain age or living in the region where the event is organized. Here too, eID cards can help in securely conveying (proving) that these conditions are satisfied for the buyer. However, the use of these cards will usually disclose more information than is required.

For big events with thousands of attendants, the police would be helped if tickets were not anonymous, but could be linked to the identity of the attendants, or at least to the identity of the buyers of these tickets. Especially, when riots occur, it would make it easier to identify and prosecute the instigators. However, the use of tickets attributable to individuals poses severe privacy risks and brings us closer to a "Big Brother" state.

This section proposes two solutions where the eID card is needed to obtain an anonymized permit, allowing a user to obtain tickets in a privacy friendly way. The role of the eID card is thus reduced to a bootstrapping role. A first solution is based on pseudonym certificates, i.e. X.509 certificates containing a user's nym instead of a real identity. A second solution is based on the more enhanced anonymous credential systems, which allow to anonymously disclose only a subset of the personal attributes (or properties thereof) embedded in the credential. Both solutions are validated and compared with the trivial solution and with each other.

We start by listing the mailn reletad work in section A.2. The main requirements are

given in section A.3. Section A.4 explains notations and specifies the assumptions. Sections A.5, A.6 and A.7, discuss the trivial protocol and two privacy friendly alternatives and are followed by a comparison in section A.8.

A.2 Related Work

Ticketing framework [21], hybrid electronic ticketing [34] and ticket for mobile user and communication [9] [28] are valuable contributions for building future ticketing systems. However, except for [34], all fall short in properly addressing user privacy. In comparison, we propose two solutions that preserve the user's privacy and avoid arbitrary blacklisting.

Heydt-Benjamin et al.[34] propose a hybrid electronic ticketing system which uses passive RFID transponders and higher powered computing devices such as smart phones or PDAs. Their hybrid ticketing system framework takes the advantage of e-cash, anonymous credentials and proxy re-encryption[22] to alleviate the concern of privacy in public transportation ticketing systems.

In general, anonymous credential protocols as described in [13], [12] commonly use a Trusted Third Party (TTP) to selectively deanonymize (or link) misbehaving users. However, Patrick et al. [27] strongly argued that deanonymizing a user with the help of TTP is a too heavy measure against a misbehaving user in a privacy-preserving system. Some applications might not necessarily need deanonymization to discourage misbehaving users, they can simply blacklist user pseudonyms, to block a user without actually revealing that user's identity. Thus, the authors propose a scheme where user misbehaviour is judged *subjectively* and blacklisted by each individual service provider (SP) without the need for TTP. Although subjective blacklisting reduces the size of a blacklist in comparison with the usual centralized blacklisting approach, it can empower a SP to arbitrarily discriminate (or freely blacklist) among its ticket users. In comparison, our protocols do not allow SPs to blacklist a user or to maintain its own blacklist. As discussed previously, in our protocols the blacklist is centrally managed by a *trusted* government instance and forwarded to the SPs. Moreover, arbitrary user blacklisting is forbidden without a judicial verdict.

A.3 Requirements

The protocols that are discussed will be evaluated based on the following requirements. F4 and F5 are optional.

Functional/Security Requirements

F1 Every event may have a policy that limits the number of tickets obtainable by one buyer. The limit may depend on a property of the buyer.

F2 Event organizers may choose to offer a subscription for a series of events.

F3 Every event can have a pricing policy that differentiates between different groups of buyers (e.g. youngsters or elderly people).

(F4) When abuse is detected or when serious incidents happen during the event, it should be possible to identify the buyer of the ticket(s) involved.

(F5) Individuals who have been imposed a banning order for a particular event type, should not be able to buy tickets for this kind of events.

Privacy Requirements

P1 Buyers of tickets should not directly be identifiable.

P2 Except when subscriptions are used, it should not be possible to compile buyer's profiles.

P3 It should not be possible to identify an individual on a blacklist.

A.4 Assumptions and Notation

The general assumptions and notation w.r.t. the protocols are now summed up.

A.4.1 Assumptions

- For every protocol, a server always first authenticates to U using a classical X.509 certificate. Also, an integrity and confidentiality preserving connection is established during a protocol. Anonymity at the network layer is added when necessary.
- A ticketing server can service multiple events. However, for each event, there is only one ticketing server.
- Tickets do only contain a ticket identifier (e.g. event name, date and seat number) and are unforgeable.

A.4.2 Notation

- Each protocol requires the following roles: user U (client), ticket server T (issues tickets to users), event organizer E and the court of justice J.
- $U \leftrightarrows B \leftrightarrows T$: $(\text{PayProof}_\text{U}, \text{PayProof}_\text{T}) \leftarrow \mathsf{pay}(\mathit{price}, \mathit{Msg})$. U pays an amount of money, via an intermediary bank B, to T. A message can be linked to the payment. The bank can deliver proofs of the payment to both parties. The payment protocols can preserve U's privacy.
- $U \leftrightarrows T$: $(\mathit{desc}[], \mathit{price}, [\mathit{Proof}]) \leftarrow \mathsf{negotiate}(\mathit{cert} \vee \mathit{Cred}, \mathit{Nym} \vee \mathit{Id}, \mathit{event}, \mathit{eventPolicy}, \#\mathit{tickets}, \mathit{specification})$ allows U and T to agree on the exact seat numbers as well as on the total price. Therefore, U gives an identifier (*Nym* or *Id*), shows (properties of) credential/certificate attributes. The event policy can state e.g. that people younger than 18 get reductions. Evidently, the number and (general) specification of the tickets are given as well. The restrictions on the blacklists can further constrain the possibilities of the user. U can give T a proof of the agreement (signed by *cert* or *Cred*).
- O: $\text{Nym} \leftarrow \mathsf{retrieveOrGenerateNym}(\mathit{Id} \vee \text{Nym}')$ returns a newly generated nym if the user referred to by *Id* or Nym' does not yet have a nym with O. However, if that user already has been given a nym in the past, it is simply retrieved from the local storage system.

- T: *Restrictions* ← retrieveRestrictions(*Blacklist*, *Nym* ∨ *Id*). The ticketing service looks up the restrictions of a *Nym* or *Id* in a blacklist.

- G: Restriction[] ← getRestrictionBooleans(*Id*) implicitly uses all blacklists, and returns for each event type whether or not the user is blacklisted or not.

- Other, self explaining methods are: add(), lookup(), store(), update () and generateTickets().

A.5 Trivial eID-based Solution

A.5.1 Introduction

Without alternatives, this protocol will most likely be be implemented in Belgium as the government is really promoting the use of the eID card in commercial applications. However, this protocol has serious privacy drawbacks.

A.5.2 High Level Description

U uses his eID card to authenticate to T, revealing a lot of personal data to T. A government agency G maintains a blacklist containing identifiable user ids. This blacklist is checked by T before issuing tickets.

A.5.3 Protocols

The protocols are given in table A.1 and are quite self explaining. The user authenticates to T using his eID card. T first checks whether the user is blacklisted. Based on the user's id and personal attributes, the user can be given the possibility to buy a number of tickets as a result of the negotiation phase. After the payment and ticket issuance, T finally stores ticket selling info.

Dispute handling is straight forward: since T knows the *link* between the seat (or ticket) and the user's id.

A.5.4 Evaluation

The functional/security requirements are trivially fulfilled. However for the privacy requirements, this protocol fails completely. T knows the user's id and all other attributes contained in the eID certificate (P1). User profiling is trivial for T as well as sharing and linking of profiles (P2). The users' ids are on the blacklist (P3). In addition, many countries simply forbid blacklists on which users are identifiable due to privacy legislation. Deployment will often thus result in omitting the F5 requirement.

(A.1.a) Getting a ticket

(1) U → T : authenticate(*eID*)

(2) T : *Restrictions*← getRestriction(*eID.NRN*, Blacklist[EventType])

(3) U ⇆ T : (SeatNB[], Price) ← negotiate(*eID*, eventPolicy, Restriction[])

(4) U, T : if (SeatNb[] = ∅) abort

(5) U ⇆ B: pay(price, H(SeatNR[], ...))
⇆ T

(6) U ← T : tickets[] ← generateTickets(SeatNb[])

(7) T : update [*eID.NRN*, event, tickets[]]

(A.1.b) Maintaining the blacklists

(1) J → G : *eID.NRN*, Restrictions, eventType

(2) G : Blacklists[EventType].add(*eID.NRN*, Restrictions)

(3) G → T : Blacklists

Table A.1: Protocols in trivial implementation

A.6 Solution based on Pseudonym Certificates

A.6.1 Introduction

This approach improves the privacy of the user by using pseudonymous permits. A unique pseudonymous root certificate is issued by the government. This allows the user to obtain pseudonymous permit certificates from different permit servers. One permit server could for instance be responsible for one event type (e.g. soccer matches). With such a pseudonymous permit a user can buy tickets for events that happen in a small (permit specific) time period[1]. The user will thus most likely need multiple permits. The blacklists no longer contain user identifiers, but pseudonyms.

A.6.2 Roles

Besides the already defined U, T and E, a government agency G is needed to issue root certificates and a permit server PS issues permit certificates.

A.6.3 Assumptions

- All certificates contain a unique serial number, a pseudonym or id, a public key and an expiry date.
- There are many pseudonym servers (PS) and many ticket servers (T).
- For every event, the ticket server (T) accepts permits. issued by a limited set of pseudonym servers. However, the user sets of different pseudonym servers do not overlap (necessary for requirement F1).

[1] The fixed time period is introduced to minimize linkabilities.

- Only one entity G can issue valid pseudonymous root certificates.
- Nyms that are no longer valid, are forgotten by the permit server.

High Level Description and data structures. The user receives a pseudonymous root certificate (Cert^{R}), which contains a rootnym (Nym^{R}) and possibly other attributes (such as year of birth, citizenship, place of residency, ...). Cert^{R} is used to authenticate to the permit server PS.
The user can apply to the PS for a pseudonym (Nym^{P}) that is valid during a predefined time period. Nym^{P} will be certified in a (pseudonymous) permit certificate (Cert^{P}). Each certificate also contains a public key used to verify authentications and signatures with Cert^{P}, and possibly (properties of) other attributes that were copied from the root certificate (Cert^{R}). Using permit certificates with non-overlapping time-slots, each user can have at most one valid Cert^{P} to order tickets for a particular event. The PS can refuse permits to users who have been sentenced to a banning order for events supported by the PS.

A.6.4 Protocols

Getting a root certificate. A governmental instance G issues a single pseudonymous root certificate Cert^{R} to each citizen. This Cert^{R} contains a pseudonym Nym^{R} that was either newly generated or retrieved by G in case the user was already assigned a Nym^{R} in the past. The user can request G to copy (properties of) attributes from in his *eID* card into his Cert^{R}[2]. G finally stores the user's *NRN* and Cert^{R}s (which include Nym^{R}.)

Getting a permit certificate. U authenticates with a valid root certificate Cert^{R} to the PS. PS will issue a number of permit certificates Cert^{P}s which have to be used before a (user specified) date (validThru). For instance, the user can request permit certificates that allow him to buy soccer tickets for the upcoming year. PS generates a set of nyms (Nym^{R}) or retrieves them (if they were already assigned in the past): one nym per time period[3]. Each nym Nym^{P} is also certified in a permit certificate Cert^{P} which also contains a validity period (for Nym^{P}), possibly a set of attributes and an encryption of the user's root pseudonym Nym^{R}. The validity periods of Nym^{P}s are non-overlapping. Hence, users cannot buy tickets for the same event using different nyms. Also, when a user requests a new permit for the same period (e.g. because the previous one was lost or the private key was stolen), PS will always use the same nym (Nym^{P}). Each Cert^{P} contains a probabilistic encryption of Nym^{R} with the public key of PS. This allows to enforce certain control measures in case of abusive behaviour (see further). PS finally updates the list of Cert^{P}s that are issued to Nym^{R}. PS can derive the time intervals for which a Nym^{R} has obtained a valid Cert^{P} from that list.

Buying tickets for an event. The user first authenticates to the ticket server T using the permit certificate Cert^{P} that is valid for that specific event and specifies the number of tickets

[2] The user can request several root certificates, each including a different set of properties/attributes. However, all certificates will refer to the same Nym^{R}.

[3] The length of the non-overlapping time periods is chosen by the PS in such a way that the number of events that fall in each period is limited.

he wants to order. T then verifies whether the pseudonym Nym^{P} is blacklisted for that event. T also checks whether Nym^{P} can still order the requested number of tickets for that event. If both conditions are fulfilled, the user and the ticket server agree on the price of the tickets and the seats. The price can depend on certain attributes that are embedded in the permit certificate (such as the user's age). After payment, the user retrieves the tickets. Finally, the ticket server updates the number of tickets that are sold to Nym^{P} for that event.

Updating anonymous blacklists. To fulfil requirement F4, anonymous blacklists are used. Four entities are involved in updating blacklists (see table A.3).
A law enforcement entity J forwards the court orders (NRN, banning order) to G. G substitutes the NRNs with the corresponding Nym^{R}s and forwards the list to the permit server PS. PS can then add Nym^{R} to a blacklist for certain event types (i.e. PS will no longer issue Cert^{P}s to Nym^{R} for the event types that are specified in the blacklist).
Finally, PS retrieves the valid Nym^{P}s for each Nym^{R} with a banning order, substitutes every Nym^{R}-record in the blacklist with a number of Nym^{P}-records and forwards the new list to the ticket server T. T no longer issues tickets to pseudonyms in the blacklist. Note that the ticket service can even revoke tickets that were already issued to pseudonyms in the blacklist.

Identifying buyer of a ticket To reveal the identity of a participant with a specified seat number, the ticket service T looks up the Nym^{P} of the user that ordered the ticket. The corresponding permit certificate Cert^{P} is kept by the ticket server and is passed to pseudonym server. The latter can link Cert^{P} to Nym^{R} (as Nym^{R} is encrypted with the public key of the pseudonym server in Cert^{P}). G can reveal the user behind Nym^{R} (as G knows the mapping between NRN and Nym^{R}). Note that a law enforcement entity J typically intermediates in the deanonymization procedure (see table A.3).

A.6.5 Evaluation

F1 This requirement is easily fulfilled as each user has only one $\text{Nym}^{\text{P}}$to buy tickets for a particular event.

F2 If Nym^{P} can be used to order tickets for multiple events (such as multiple soccer games during a World Cup contest), T can even restrict the total number of tickets that can be bought for the whole contest (i.e. a set of events).

F3 A user can get a lower price for some tickets based on the attribute values of Cert^{P}. However, tickets can be passed on. Hence, T should be careful with price reductions.

F4 Fulfilled (cfr. *"Identifying buyer of a ticket"* protocol).

F5 Three entities are needed to ban a user from event types for which a user already has a permit certificate, namely G, PS and T. Two entities are needed to ban a user from event types for which a user does not yet have a permit certificate, namely G and PS.

P1 As discussed in *"Identifying buyer of a ticket"*, four entities are needed to reveal the user's identity. Moreover, G (and maybe PS) are governmental instances. Hence, users can trust that players in the commercial sector (such as E and T) cannot identify users without help of governmental instances.

(A.2.a) Getting a pseudonymous root certificate Cert^{R}

(1) U $\rightarrow$ G : authenticate(eID)

(2) G : $\text{Nym}^{\text{R}} \leftarrow$ retrieveOrGenerateNym($eID.NRN$)

(3) U $\leftarrow$ G : $\text{Cert}^{\text{R}} \leftarrow$ issueCert({Nym^{R}, attributes ...})

(4) G : store[$eID.NRN$, Cert^{R}]

(A.2.b) Getting a permit certificate Cert^{P}

(1) U $\rightarrow$ PS : authenticate(Cert^{R})

(2) U $\rightarrow$ PS : validThru, *attributes to include*

(3) PS : $\forall$ [from,till], from $\leq$ validThru:

(4) PS : $\text{Nym}^{\text{P}} \leftarrow$ retrieveOrGenerateNym($\text{Cert}^{\text{R}}.\text{Nym}^{\text{R}}$, [from,till])

(5) U $\leftarrow$ PS : $\text{Cert}^{\text{P}} \leftarrow$ issueCert(Nym^{P}, [from,till], *attributes* $\text{enc}_{pk_{SP}}$(RAND || $\text{Cert}^{\text{R}}.\text{Nym}^{\text{R}}$),)

(6) PS : store[$\text{Cert}^{\text{R}}.\text{Nym}^{\text{R}}$. [from,till], Cert^{P}]

(A.2.c) Buying tickets

(1) U $\rightarrow$ T : authenticate(Cert^{P})

(2) U $\rightarrow$ T : event, #tickets, specification

(3) T : *Restrictions*$\leftarrow$ retrieveRestrictions($\text{Cert}^{\text{P}}.\text{Nym}^{\text{P}}$, EventType)

(4) U $\leftrightarrows$ T : (SeatNb[], price) $\leftarrow$ negotiate($\text{Cert}^{\text{P}}.\text{Nym}^{\text{P}}$, event, #tickets, eventPolicy, Cert^{P}.attr, specification, [*Restrictions*])

(5) U, T : if (SeatNb[] = $\oslash$) abort

(6) U $\leftrightarrows$ B: pay(price, Hash(SeatNb[], ...)) $\leftrightarrows$ T

(7) U $\leftarrow$ T : tickets[] $\leftarrow$ generateTickets(SeatNb[])

(8) T : update [Cert^{P}, event, tickets[]]

Table A.2: Protocols with pseudonym certificates

P2 Each Nym^{P} only has a limited validity period. The number of tickets that is issued to the same Nym^{P} is restricted. Hence, T and E can only compile limited profiles. PS can link all Nym^{P}s to the same Nym^{R}. However, multiple pseudonym servers PS can be used. If each PS can only issue permit certificates for specific types of events, the one PS cannot link multiple interests of the same Nym^{R}.

P3 Only Nym^{R}s and Nym^{P}s are kept in blacklists.

A.7 A Ticketing System Based on Anonymous Credentials

A.7.1 Introduction

We further increase the user's privacy. The user needs a single permit - issued by a government agency - which allows the user to buy tickets for every event. In case of abuse, the transcript

(A.3.a) anonymizing the blacklists

(1) J → G : [NRN, banning order eventType]

(2) G : $\text{Nym}^{\text{R}} \leftarrow \mathsf{lookupNym}(NRN)$

(3) G → PS : [Nym^{R}, banning order eventType]

(4) PS : $\text{Nym}^{\text{P}} \leftarrow \mathsf{lookupNym}(\text{Nym}^{\text{R}}, \text{eventType})$

(5) PS → T : [Nym^{P}, banning order eventType]

(A.3.b) Identifying buyer of a ticket

(1) J ← E : complaint, seatNb

(2) J → T : event, seatNb

(3) J ← T : [Cert^{P}, event, ticket] ← lookup(event, seatNb)

(4) J → PS : Cert^{P}

(5) J ← PS : (RAND || Nym^{R}) ← $\mathsf{dec}_{prk_{SP}}(\text{Cert}^{\text{P}}.\text{enc})$

(6) J → G : Nym^{R}

(7) J ← G : $NRN \leftarrow \mathsf{lookup}(\text{Nym}^{\text{R}})$

Table A.3: Protocols with pseudonym certificates (bis)

resulting from the permit show can be deanonymized. For each event type, there is a privacy-preserving blacklist, summing up the user's rights restrictions.

A.7.2 Roles

Besides U, E, T, and J, we define G as a government agency that issues permits and manages blacklists.

A.7.3 Assumptions

In the ticketing system based on anonymous credentials, we assume the following:

- The anonymous credential system provides the unlinkability property to permits. The user does not reveal identifiable permit attribute properties.

- All Es and all Ts and G have a unique, publicly available provable one-way function; $\text{f}^{\text{E}}()$ for E, $\text{f}^{\text{T}}()$ for T) and $\text{f}^{\text{G}}(.\ ,\ .)$ for G. Note that the latter requires two arguments. These functions could for instance be included in their X.509 certificate.

- The opening info generated by a commit method does not reveal any information about the content contained in the commitment. This is easily achieved using a symmetric key K:
  $Com^{new} \leftarrow (Com,\ enc_K(\text{OpenInfo}))$ $\text{OpenInfo}^{new} \leftarrow K$ combined with integrity preserving measures (e.g. MACs).

A.7.4 High Level Description

The permit is an anonymous credential containing a set of personal attributes, a boolean value for each event type indicating whether or not the user is blacklisted, and two nyms. One nym (Nym^{R}) is known by G and used to blacklist persons. The other nym (Nym^{P}), is not known to G, but is used to generate an event specific nym, allowing T to keep track of the number of tickets sold to that person for that specific event.

Per event type, a blacklist is maintained by G. This blacklists contains user pseudonyms (Nym^{R}s). These nyms are converted to event specific nyms (Nym^{E}s) before the blacklist is sent to a specific T as a way to avoid linkabilities.

A.7.5 Protocols

Getting an anonymous Permit Certificate. The actual issue of the permit (A.4.a.5) includes a subset of the user's personal attributes (*attributes*) contained in the user's eID. These can be selectively disclosed during a credential show protocol.

The permit contains for each event type a boolean *Restrictions*[EventType] stating whether or not the user is blacklisted. G can easily extract this information out of the blacklists it manages (cfr. below).

Each permit contains two user unique pseudonyms Nym^{R} and Nym^{P}. Nym^{R} is known to both U and G and is the nym under which the permit is issued by G. G possesses a provable link Sig^{R} between the U's id and his Nym^{R}. This can be used in case of disputes.

The second pseudonym in the permit, Nym^{P}, is known to the user U only and is included in the permit as an attribute that is not known to G. This is done using a commitment, whereof U proves that he knows the corresponding *UserSecret* and Nym^{P} (underlined) such that $\text{Nym}^{\text{P}} \leftarrow \text{f}^{\text{G}}(\text{Nym}^{\text{R}}, \textit{UserSecret})$.

To obtain a new permit, after the previous one was lost, step 6 changes. After recalculating $\text{Nym}^{\text{P}} \leftarrow \text{f}^{\text{G}}(\text{Nym}^{\text{R}}, \textit{UserSecret})$ and generating a new commitment $\textit{Com}2 \leftarrow \mathsf{commit}(\text{Nym}^{\text{P}})$ (Step 4 and 5), U decrypts c, resulting in the opening info of the previous commitment. This allows U to prove that $\textit{Com}.\text{Nym}^{\text{P}} = \textit{Com}2.\text{Nym}^{\text{P}}$ (corresponds to step 6), convincing G that the same Nym^{P} was used.

Buying a Ticket. For each ticket order, U sends $\text{Nym}^{\text{E}} \leftarrow \text{f}^{\text{E}}(\text{Nym}^{\text{P}})$ to T and proves possession of the corresponding Nym^{P}. (A.4.c.1,2). The use of one-way function gives the user for each event a different, but event-unique nym. This gives T the possibility to limit the number of tickets per user while at the same time, this function avoids linking of T's customers to the customers of other Ts. Collusion with G does not help, because G does not even know Nym^{P}.

When ordering a ticket, the user proves that he is not blacklisted by showing *Restrictions*[EventType]. If U is blacklisted, he sends $\text{Nym}^{\text{T}} \leftarrow \text{f}^{\text{T}}(\text{Nym}^{\text{R}})$ to T and proves that Nym^{T} is correctly formed with $\text{Cred}^{\text{P}}.\text{Nym}^{\text{R}}$. T now looks up the exact restrictions associated with Nym^{T} on the blacklist (A.4.c.3). This limits linking possibilities and possible collusion with G. The latter is only useful for blacklisted Us.

The negotiation phase (A.4.c.4) requires the user's permit as input, such that RequestProof can be generated. RequestProof is a proof for G that U did request the negotiated tickets at the negotiated price. This proof is also deanonymizable by J which provably reveals Nym^{R}.

(A.4.a) Getting the first anonymous permit certificate $\mathrm{Cred}^{\mathrm{P}}$

(1)	$\mathrm{U} \rightarrow \mathrm{G}$	:	$\mathsf{authenticate}(eID)$
(2)	$\mathrm{G} \leftrightarrows \mathrm{U}$	:	$(\mathrm{Nym}^{\mathrm{R}}, \mathrm{Sig}^{\mathrm{R}}) \leftarrow \mathsf{generateSignedNym}(eID.NRN)$
(3)	G	:	$\mathrm{Restriction}[] \leftarrow \mathsf{getRestrictionBooleans}(\mathrm{eID}.NRN)$
(4)	$\mathbf{U} \leftrightarrows \mathrm{G}$	:	$\mathrm{Nym}^{\mathrm{P}} \leftarrow \mathrm{f}^{\mathrm{G}}(\mathrm{Nym}^{\mathrm{R}}, UserSecret)$
(5)	$\mathrm{U} \rightarrow \mathrm{G}$	:	$(Com, \mathrm{OpenInfo}) \leftarrow \mathrm{Comm}(\mathrm{Nym}^{\mathrm{P}})$
(6)	$\mathrm{U} \rightarrow \mathrm{G}$	:	$Com, \mathsf{prove}(\underline{Com.\mathrm{Nym}^{\mathrm{P}}} = \mathrm{f}^{\mathrm{G}}(\mathrm{Nym}^{\mathrm{R}}, \underline{UserSecret})),$ $c \leftarrow enc_{H(UserSecret)}(\mathrm{OpenInfo})$
(7)	$\mathrm{U} \leftrightarrows \mathbf{G}$	:	$\mathrm{Cred}^{\mathrm{P}} \leftarrow \mathsf{issueCred}(\mathrm{Nym}^{\mathrm{R}}, \{Com.\mathrm{Nym}^{\mathrm{P}},$ $\mathrm{Restriction}[], attributes\})$
(8)	G	:	$\mathsf{store}[eID.NRN, \mathrm{Nym}^{\mathrm{R}}, \mathrm{Sig}^{\mathrm{R}}, Com, c]$

(A.4.a) Buying tickets

(1)	$\mathrm{U} \rightarrow \mathrm{T}$	:	$\mathrm{Nym}^{\mathrm{E}} \leftarrow \mathrm{f}^{\mathrm{E}}(\mathrm{Cred}^{\mathrm{P}}.\mathrm{Nym}^{\mathrm{P}})$, event
(2)	$\mathrm{U} \rightarrow \mathrm{T}$	:	$\mathsf{authenticate}(\mathrm{Cred}^{\mathrm{P}}, \{\mathrm{Cred}^{\mathrm{P}}.\mathrm{Nym}^{\mathrm{P}} \simeq \mathrm{Nym}^{\mathrm{E}},$ $\mathrm{Cred}^{\mathrm{P}}.\mathrm{Restriction}[\mathrm{EventType}]\})$
(3)	T	:	$\mathrm{if}(\mathrm{Cred}^{\mathrm{P}}.\mathrm{Restriction}[\mathrm{EventType}] = \mathrm{true})$ do
(3.a)	$\mathrm{U} \rightarrow \mathrm{T}$	:	$\mathrm{Nym}^{\mathrm{T}} \leftarrow \mathrm{f}^{\mathrm{T}}(\mathrm{Cred}^{\mathrm{P}}.\mathrm{Nym}^{\mathrm{R}})$
(3.b)	$\mathrm{U} \rightarrow \mathrm{T}$	:	$\mathsf{prove}(\mathrm{Nym}^{\mathrm{T}} \simeq \mathrm{Cred}^{\mathrm{P}}.\mathrm{Nym}^{\mathrm{R}})$
(3.c)	T	:	$Restrictions \leftarrow \mathsf{retrieveRestrictions}(\mathrm{Blacklist}_{\mathrm{T}}, \mathrm{Nym}^{\mathrm{T}})$
(3.d)	T	:	end if
(4)	$\mathrm{U} \leftrightarrows \mathrm{T}$	:	$(\mathrm{SeatNb}[], \mathrm{price}, \mathrm{RequestProof}) \leftarrow \mathsf{negotiate}(\mathrm{Cred}^{\mathrm{P}}, \mathrm{event};$ $\mathrm{Nym}^{\mathrm{E}}, \mathrm{eventPolicy}, [Restrictions])$
(5)	$\mathrm{U} \leftrightarrows \mathrm{B} \leftrightarrows \mathrm{T}$	:	$(\mathrm{PayProof}_{\mathrm{U}}, \mathrm{PayProof}_{\mathrm{T}}) \leftarrow \mathsf{pay}(\mathrm{price}, \mathrm{Hash}(\mathrm{SeatNb}[], \ldots))$
(6)	$\mathrm{U} \leftarrow \mathrm{T}$	:	$\mathrm{tickets}[] \leftarrow \mathsf{generateTickets}(\mathrm{SeatNb}[])$
(7)	T	:	update [event, $\mathrm{Nym}^{\mathrm{E}}$, RequestProof, tickets[]]

Table A.4: Protocols with anonymous credentials

Blacklist Maintenance and Retrieval. A law enforcement entity J forwards the court orders (NRN, $Restrictions$) to G. G substitutes the NRNs with the corresponding $\mathrm{Nym}^{\mathrm{R}}$s. Each $\mathrm{Nym}^{\mathrm{R}}$is further converted to $\mathrm{Nym}^{\mathrm{T}} \leftarrow \mathrm{f}^{\mathrm{T}}(\mathrm{Nym}^{\mathrm{R}})$ before the blacklist is sent to a specific T to avoid linkabilities and profiling by T (A.5.b).

Misbehaviour and Deanonymization Protocol A.5.c illustrates how the collaboration of E, T and G is required in order to obtain a (provable) link between the ticket and the user's id. The proof is (RequestProof, deanProof, $\mathrm{Sig}^{\mathrm{R}}$). If someone is put on a blacklist for EventType, his permit $\mathrm{Cred}^{\mathrm{P}}$ is revoked. U can obtain a new $\mathrm{Cred}^{\mathrm{P}}$, with the updated restrictions booleans Restriction[EventType], immediately.

A.7.6 Evaluation

We now evaluate by checking the requirements

(A.5.a) Maintaining the blacklists

(1) J → G : Nym^{R}, Restrictions, EventType

(2) G : Blacklists[EventType].add(Nym^{R}, *Restrictions*)

(3) J → G : revokeCert(Nym^{R})

(A.5.b) Obtaining a blacklist

(1) G : for each (Nym^{R}, *Restrictions*) in Blacklists[EventType]:
$\text{Blacklist}_{\text{T}}$.add($\text{f}^{\text{T}}(\text{Nym}^{\text{R}})$, *Restrictions*)

(2) T ← G : $\text{Blacklist}_{\text{T}}$

(A.5.c) Identifying buyer of a ticket

(1) J ← E : complaint, seatNb

(2) J → T : event, seatNb

(3) J ← T : RequestProof← lookup(event, seatNb)

(4) J : Nym^{R}, deanProof ← deanonymize(RequestProof)

(5) J → G : (*NRN*, Sig^{R}) ← lookup(Nym^{R})

Table A.5: Protocols with anonymous credentials (bis)

Functional and Security Evaluation

F1 $\text{Nym}^{\text{E}}\leftarrow \text{f}^{\text{E}}(\text{Nym}^{\text{P}})$ enables T to link ticket orders of the same U for the same event.

F2 A subscription can be issued by T or a coordinating organization. It can be an anonymous credential that contains Nym^{P}, Nym^{R}, the Restriction[EventType] booleans and information about the subscription. It can be anonymously shown to a ticketing service in order to obtain tickets without a payment phase. Alternatively, a multiple-use ticket with an expiry date can be issued.

F3 The user can selectively disclose properties in the permit.

F4 is explained in section A.7.5.

F5 is done using the anonymized blacklists. Revocation of tickets issued to persons that were blacklisted after the ticket order is possible if Nym^{R} is systematically shown to T. However, the price is an increase in linkabilities.

Privacy Evaluation

P1 Deanonymization requires the collaboration of T, G and J as we argued in *Misbehaviour and Deanonymization*.

P2 We argued that a user has for each E a different $\text{Nym}^{\text{E}}\leftarrow \text{f}^{\text{E}}(\text{Nym}^{\text{P}})$. Nym^{P} is needed to do linking to other Es, but can only be obtained if both the user's secret *UserSecret* and the user's Nym^{R} are known. For blacklisted users, G can link Nym^{R} and Nym^{T}. Collusion of T and G is then possible.

P3 G knows the links between nyms on a blacklist and the user's id. However, such convictions are publicly available. Collusion of T and G can reveal the identity associated with Nym^{T}.

	Trivial	**Pseudonym certs.**	**Anon. creds.**
F1 - # Tickets	✓	✓	✓
F2 - Subscription	✓	✓	✓
F3 - Pricing	✓	✓	✓
F4 - Deanon.	✓	✓- J interacts with E, T, PS, G.	✓- J interacts with E, T, G.
F5 - Ban	—	✓+ ticket revocability	✓(2)
P1 - User anon.	T knows user id	If no collusion of E, T, PS, G. T knows permit atts.	✓
P2 - User profiles	T can link everything.	Linkability during limited, fixed period.	✓(1)
P3 - Anon. blacklists	—	If no collusion PS, G.	only G can identify. U.

(1): If the user is blacklisted, G can collude with one or more Ts.

(2): Ticket revocability is possible at the cost of increased linkabilites.

Table A.6: Comparison of the three approaches

A.8 Evaluation

A comparison of the three approaches is given in table A.6. It is clearly possible to fulfil the main functional/security requirements, while at the same time giving the privacy a serious boost. To maintain user-friendliness, the interactions with e.g. PS can be done transparently to the user. The anonymous credential based protocol is computationally the most intensive. Tests are needed to quantify this.

We have to be aware that the two proposed solutions disallow a banned person to buy tickets for someone else (e.g. father buys tickets for his children) and that it is still possible that a person buys tickets and gives them to a banned person. The solution is thus not perfect and still, police presence is needed on e.g. soccer matches.

A.9 Conclusions and Future Work

Two privacy preserving ticketing systems were proposed; one based on pseudonym certificates and one on anonymous credentials. We showed that it is possible to offer the user a high degree of privacy, while the other requirements remain fullfilled. Still the privacy unfriendly eID card is used as bootstrap.

A prototype implementation will be made, using an applet for registration and ticket ordering. Entering the event can be done using a bar code reader. The influence of mix networks on the overall performance must be examined.

Appendix B

ePoll

B.1 Introduction

In a poll, opinions of people are collected and processed. In paper-based polls the collection and processing takes a lot of time and effort. Electronic poll systems (ePoll), however, offer several benefits with respect to the paper-based polls. ePolls enable users to sign polls anywhere at any time and now reach wider sections of society. Moreover, automatic processing of the results can make the polls more reliable.

On the other hand, electronic poll systems introduce some new problems. Some systems are unreliable and may return incorrect results as for instance a user may sign a poll more than once. Other systems, use personal information to prevent multiple signing. However these systems are not privacy friendly.

This section presents PetAnon, a privacy-preserving poll system using Idemix anonymous credentials. PetAnon combines good privacy properties with reliable results.

This appendix is structured as follows. Section B.2 presents the main related work. Section B.3 gives the requirements of the system. Section B.4 discusses the protocol used in PetAnon and is followed by section B.5 with an evaluation of the protocol in respect of the requirements.

B.2 Related Work

Many voting protocols that guarantee anonymity of the voter have been divised. A distinction needs to be made between voting protocols that require a physical appearance of the user in a booth and online voting protocols. We are interested in online protocols ([5], [2], [16], etc.). In pure voting protocols, not all the flexibility described in the introduction is required. On the other hand, secure voting protocols need to be receipt-free, which is a requirement that is not fullfilled by all voting schemes presented in the past.

Recently, an e-Petition implementation based on the Belgian eID card and Idemix was developed [10], which presents similarities with our ePoll solution, but it is less flexible and reliable. For instance, user's cannot prove that their vote was removed. In fact, we focus on a flexible and reliable poll infrastructure and its implications on anonymity and policy aspects, while their focus was presenting a proof of concept implementation of an anonymous petition system combined with a discussion about the legal issues.

B.3 Requirements

The requirements of the privacy-preserving ePetition system are discussed below. They are classified according to security and privacy requirements.

Security requirements

S1 A user can sign a certain petition only once.

S2 A petition may possibly address only a subset of the potential signers; therefore the signer may be required to prove that he belongs to that subset.

S3 A user can verify that his signature is included in the petition's database.

S4 Everyone can verify the correctness of the petition results.

Privacy requirements

P1 Signers are anonymous.

P2 Signatures cannot be linked to a user. Moreover, signatures of different petitions cannot be linked to each other.

P3 A petition may request optional attributes that the user can release in order to get more differentiated results. The user has the choice if he wants to disclose these attributes or not.

B.4 Protocols

Roles and setting. A user U possesses an eID card, which is used when U authenticates towards the registration server R. This authentication is required before R issues a *voting credential* to U that can be used to sign an ePetition on a petition server P. The registration server R has a certificate containing the public key information used in the credential-issue and credential-show protocols.

Setting up an ePetition. The petition organizer P contacts R and offers to R the title, description and validity period of the petition. R generates a new, unique provable one-way function $f_{petition}(.,.)$ which needs two arguments. This function, as well as the user provided petition info are included in a (X.509) *petition certificate* $cert_{petition}$ that is issued by R to P. As a result, the latter obtains a corresponding private key $PK_{petition}$.

Retrieving a signPetitions credential. In order to sign a petition, U has to obtain a signPetitions credential. Therefore, he authenticates using his eID card (1). This actions reveals the personal data contained in the eID card to R.

Every citizen is only allowed to have one signPetitions credential. This is first checked by R. If the user did not register beforehand (2a), the user generates a (long) secure random number (2a.1), puts it in a commitment (2a.2), which is sent to R, and proves that he knows the committed value (2a.3). R also generates a (potentially shorter) random value (2a.4).

These two random values will be used to prevent voting multiple times for the same petition, as we will see later.

If U previously had been issued a signPetitions credential (2b), these two random values are retrieved from R's storage and will be reused (2b.2). Also the credential's serial number is retrieved, which allows to revoke this credential before issuing a new one (2b.1).

After a serial number for the new credential is generated (3), all the parameters for the credential issuance are known and the signPetitions credential is issued (4). It contains the two random values, the serial number and a subset of the attributes (or properties thereof) that were extracted from the eID card. Note that R never gets hold of the user's secure random number.

Finally, U stores the credential (6), and R stores the commitment, the other random number and the serial number, as well as the user's NRN (National Registration Number) (5). This will allow R to check whether a user already has been issued a signPetitions credential, to revoke signPetitions credentials and to issue new ones.

Signing a petition. Initially, the petition server P authenticates using his petition specific certificate $cert_{petition}$. P additionally sends an overview of required and optional personal properties that must or can be proved when signing the credential. Each petition has certain required and optional attributes. For instance to sign a certain petition you must be older than 18 years. However, it is up to the user if he wants to reveal his gender or zip code. Finally, P sends a list of options for which the user can vote to U (1).

With the help of the petition's one-way function and the two random values contained in the vote credential, the user generates his petition specific nym (2), and sends it to P, together with the description of the personal properties that U is willing to disclose and the option for which he wants to vote (3).

Now, the interactive Idemix show protocol is run (4): U proves the selected properties, as well as that the petition specific nym for that user is correctly formed based on the random values contained in the credential. Thereby the user's vote choice is anonymously signed.

If that nym has not yet signed that specific petition (5), the protocol continues by generating a vote number. This is a reference to the petition-record that is being generated. The vote number, the hash of the proof and the user's nym are signed with the petition secret key, and stored by P together with that signature. The resulting record is made public. The signature is sent to and stored by U and allows U to check that his signature is included in the petition's database and to file a complaint otherwise and proof whether the record is changed by P.

Verification. The user can request from P the record with index voteNrand which was signed by the user. If the vote was tampered with, either the P-provided signature will no longer equal the signature stored by U, or the P-provided signature will no longer match the $(proof, nym, voteNr)$-tuple made public by P.

If all the records are made publicly available, everyone can verify the correctness of the petition by verifying for each record the proof and the respective signature.

(B.1.a) Retrieving an anonymous credential

(1)	U → R	:	authenticate(eID)
(2a)	R	:	if (!credExists($eID.NRN$))
(2a.1)	U	:	secureRand← genSecureRand()
(2a.2)	U	:	(Comm, OpenInfo) ← commit(secureRand)
(2a.3)	U → R	:	Comm, prove({x | Comm== commit(x)}, Comm,OpenInfo)
(2a.4)	U → R	:	rand_R← genRand
(2b)	R	:	else
(2b.1)	R	:	(serialOld, Comm, rand_R) ← retrieveCredInfo($eID.NRN$)
(2b.2)	R	:	revokeCred(serialOld)
(3)	R	:	serial← genSerial()
(4)	U ⇆ **R**	:	Cred← issueCred(serial, Comm.secureRand, rand_R, subset(properties_{eID}))
(5)	R	:	store(serial, $eID.NRN$, Comm, rand_R)
(6)	U	:	store(Cred)

(B.1.b) Signing petitions

(1)	U ← P	:	authenticate($cert_{petition}$), $Options_{props}$, choices[]
(2)	U	:	$\text{Nym}_{\leftarrow}$ $cert_{petition}.f_{petition}$(Cred.secureRand, Cred.serial)
(3)	U → P	:	$\text{Nym}_{,}$ $props$ ← select($Options_{props}$), $choice$ ← $select(choices[])$
(4)	**U** ⇆ P	:	proof← showCred(Cred, $props$ && $\text{Nym}_0 \backsim$ Cred){choice}
(5)	P	:	if (petitionSigned($\text{Nym}_)$ abort()
(6)	U ← P	:	voteNr← getVoteNr()
(7)	U ← P	:	receipt← sig($SK_{petition}$, (voteNr, hash(proof), Nym_0))
(8)	P	:	store[voteNr, Nym_0, proof, receipt]
(9)	U	:	store[receipt, hash(proof), voteNr]

Table B.1: Protocols for PetAnon

B.5 Evaluation

S1 is easily fulfilled, as for each petition the user is known by P under a petition specific nym. If that nym already signed that specific petition, the vote is cancelled.

S2 and [P3] are fulfilled. Some attributes in the credential show may be required by P, while others are up to U if he wants to disclose the information or not.

S3 is fulfilled. U can detect if his vote was tampered with based on the P-provided signature.

S4 If the records are made publicly, everyone can verify the correctness of the petition by verifying the proofs and signatures.

P1 Using the Idemix credential show protocol, as long as no identifying attributes are revealed, the user U remains anonymous, and different shows are unlinkable. Moreover, privacy is preserved in case of collusion of R and P.

P2 is fulfilled. To sign a petition, U authenticates anonymously using his credential (Cred). Signing the petition is done anonymously, and there are no identifiable actions linked to the signature.

Appendix C

Modelling

Class	Description
`ActorApplication`	This construct models the interacting parties.
`LifeLine`	This construct provides a visual means of ordering a party's operations over time.
`Operation`	This abstract construct represents an operation that involves one or more parties.
`SingleActorOperation`	This abstract construct represents an operation that involves a single party. For instance, data input by the user.
`InterActorOperation`	This abstract construct represents an operation that involves two or more parties. For instance, the transmission of data from a source to a destination.
`ReadData`	This construct models the gathering of user information. The said information may be referenced by other constructs by means of the inherited `id` attribute and can be transmitted or used for authentication purposes.
`SanitizeData`	This construct models the hiding of arbitrary information. For instance, a party may wish to hide certain of its attributes from some parties but not from others.
`InformationMessage`	This construct is more related to the user-interface aspect of the application that any of the others. It models the display of textual information to the user to guide him through the application.
`SendData`	An `InterActorOperation` which connects two distinct `LifeLine`s that represent the source and destination parties. This construct models the transmission of data between parties.
`ShowCredential`	An `InterActorOperation` which connects two distinct `LifeLine`s that represent the credential "shower" and credential verifier. This construct models the authentication of one party to another.

Table C.1: A description of the classes and relationships from Figure 5.3.

Class	Attribute	Description
`ActorApplication`	name	A party's descriptor.
`Operation`	id	This attribute is inherited by all children of the `Operation` construct. It enables unique referencing of one construct by another.
`ReadData`	clientChosenAttributes	A comma separated list of attributes the concerned party chose to provide.
	issuerChosenAttributes	A comma separated list of attributes the verifying party requires.
	issuerChosenIdentificationAttribute	A unique identifier attribute the verifying party requires.
	dataSource	The location from which to retrieve the data. This attribute may take values *userProvided* (indicating data will be entered by the user at runtime) or *eIDCard* (indicating the data should be read from a Belgian eID card).
	dataTitle	The read data's descriptor.
	inputPrompt	A message that should be displayed to the user to inform him that some of his data is required.
`SanitizeData`	attributesToRemove	A comma separated list of attributes to remove from the current data.
	readDataId	Identifies the `ReadData` instance whose data will be sanitized.
`InformationMessage`	exitEvent	This trigger that will prompt the application to navigate away from the displayed message. This attribute may take values *userClick* (indicating that the user will click an "Ok" button when he wishes to move on) or a numeric value in milliseconds (indicating the delay that should elapse before automatically moving on).
	message	The message to be displayed.
`SendData`	readDataId	Identifies the `ReadData` instance whose data will be transmitted to the destination party.
`ShowCredential`	credentialType	The type of credential we wish to show. This attribute may take values *publicKeyCertificate* or *anonymousCredential*.
	credentialIssuingAuthority	In the case of public key certificates, this attribute indicates the certification authority which will be contacted to verify the credential.
	predicatesToProve	A comma separated list of predicates that an anonymous credential must satisfy.
	readDataId	Identifies the `ReadData` instance whose data will be used to build the credential.
	showingMultiplicity	Indicates the possibly infinite number of times this credential can be shown before it expires. This will impact the created credential and the generated code for the following `ShowCredential` instances.

Table C.2: A description of the class attributes from Figure 5.3.

Bibliography

[1] Belgian certificate revocation list. `http://status.eid.belgium.be/`.

[2] Alessandro Acquisti. Receipt-free homomorphic elections and writein ballots. Technical report, 2004.

[3] Patrick Andries. *eID Middleware Architecture Document*. Zetes, 1.0 edition, 2003.

[4] N. Asokan, Els Van Herreweghen, and Michael Steiner. Towards a framework for handling disputes in payment systems. Technical Report RZ 2996, 1998.

[5] Josh Benaloh and Dwight Tuinstra. Receipt-free secret-ballot elections (extended abstract). In *STOC '94: Proceedings of the twenty-sixth annual ACM symposium on Theory of computing*, pages 544–553, New York, NY, USA, 1994. ACM.

[6] Jean Bezivin. On the unification power of models. *Software and Systems Modeling (SoSym)*, 4:171–188, 2005.

[7] S. Brands. A technical overview of digital credentials, 1999.

[8] Alan W. Brown. Model driven architecture: Principles and practice. *Software and Systems Modeling (SoSym)*, 3:314–327, 2004.

[9] L. Buttyn and J. P. Hubaux. Accountable anonymous access to services in mobile communication systems. In *Proceedings of the 18th IEEE Symposium on Reliable Distributed Systems*, 1999.

[10] M. Kohlweiss C. Diaz, H. Dekeyser and G. Nigusse. Privacy preserving electronic petitions, 2008.

[11] J. Camenisch and E. Van Herreweghen. Design and implementation of the idemix anonymous credential system, 2002.

[12] J. Camenisch and E.V. Herreweghen. Design and implementation of the idemix anonymous credential system. In *ACM Computer and Communication Security*. 2002.

[13] Jan Camenisch and Anna Lysyanskaya. An efficient system for non-transferable anonymous credentials with optional anonymity revocation. In *EUROCRYPT '01: Proceedings of the International Conference on the Theory and Application of Cryptographic Techniques*, pages 93–118, London, UK, 2001. Springer-Verlag.

[14] D. Chaum. Security without identification: transaction systems to make big brother obsolete. *Commun. ACM*, 28(10):1030–1044, 1985.

[15] Krzysztof Czarnecki and Simon Helsen. Feature-based survey of model transformation approaches. *IBM Systems Journal (IBMS)*, 45:621–645, 2006.

[16] Peter Y.A. Ryan David Chaum and Steve Schneider. A practical voter-verifiable election scheme. In *Computer Security ESORICS 2005, Lecture Notes in Computer Science*, pages 118–139, Berlin / Heidelberg, Germany, 2005. Springer.

[17] Danny De Cock, Christopher Wolf, and Bart Preneel. The Belgian Electronic Identity Card (Overview). In Jana Dittmann, editor, *Sicherheit 2005: Sicherheit - Schutz und Zuverlässigkeit, Beiträge der 3rd Jahrestagung des Fachbereichs Sicherheit der Gesellschaft für Informatik e.v. (GI)*, volume LNI P-77 of *Lecture Notes in Informatics (LNI)*, pages 298–301, Magdeburg,DE, 2006. Bonner Köllen Verlag.

[18] Danny De Cock, Karel Wouters, and Bart Preneel. Introduction to the Belgian EID Card: BELPIC. In Stefanos Gritzalis, Sokratis K. Katsikas, and J. Lopez, editors, *European PKI Workshop: Research and Applications*, volume 3093 of *Lecture Notes in Computer Science*, pages 1–13, Samos Island,GR, 2004. Springer-Verlag.

[19] Juan de Lara, Hans Vangheluwe, and Manuel Alfonseca. Meta-modelling and graph grammars for multi-paradigm modelling in AToM^3. *Software and Systems Modeling (SoSym)*, 3:194–209, 2004.

[20] Mohamed Fayad and Douglas C. Schmidt. Object-oriented application frameworks. *Commun. ACM*, 40(10):32–38, 1997.

[21] K. Fujimura and Y. Nakajima. General-purpose digital ticket framework. In *Proceedings of the 3rd USENIX Workshop on Electronic Commerce*, pages 177–186, 1998.

[22] M. Green G. Ateniese, K. Fu and S. Hohenberger. Improved proxy re-encryption schemes with applications to secure distributed storage. In *In: Proceedings of the 12th Annual Network and Distributed System Security Symposium (NDSS)*, 2005.

[23] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Professional, 1995.

[24] R.E. Johnson and B. Foote. Designing reusable classes. *Journal of object-oriented programming 1(2)*, pages 22–35, 1988.

[25] Steven Kelly and Juha-Pekka Tolvanen. *Domain-Specific Modeling : Enabling Full Code Generation*. Wiley-Interscience, 2008. 427 pages.

[26] Raphael Mannadiar and Hans Vangheluwe. Modular synthesis of mobile device applications from domain-specific models. Technical report, McGill University, 2010.

[27] A. Kapadia P. P. Tsang, M. H. Au and S. W. Smith. Blacklistable anonymous credentials: blocking misbehaving users without TTPs. In *CCS '07: Proceedings of the 14th ACM conference on Computer and communications security*, pages 72–81. ACM, 2007.

[28] B. Patel and J. Crowcroft. Ticket based service access for the mobile user. In *In Proceedings of Mobicom'97*, 1997.

[29] W. Pree. Meta patterns - a means for capturing the essentials of reusable object-oriented design. *in M. Tokoro and R. Pareschi (eds), Springer-Verlag, proceedings of the ECOOP, Bologna, Italy*, pages 150–162, 1994.

[30] Guy Ramlot. *eID Hierarchy and Certificate Profiles.* Zetes, Certipost, 3.1 edition, 2006.

[31] Johan Rommelaere. *Belgian Electronic Identity Card Middleware Programmers Guide.* Zetes, 1.40 edition, 2003.

[32] Laurent Safa. The making of user-interface designer a proprietary DSM tool. In *7th OOPSLA Workshop on Domain-Specific Modeling (DSM)*, page 14, http://www.dsmforum.org/events/DSM07/papers.html, 2007.

[33] Marc Stern. *Belgian Electronic Identity Card content.* Zetes, CSC, 2.2 edition, 2003.

[34] B. Defend T. S. Heydt-Benjamin, H. Chae and K. Fu. Privacy for public transportation. In *Proceedings of the Sixth Workshop on Privacy Enhancing Technologies (PET 2006).* Springer, 2006.