D4.2 Final Reference Implementation

Thomas Baignères, Patrik Bichsel, Robert R. Enderlein, Hans Knudsen, Kasper Damgård, Jonas Jensen, Gregory Neven, Janus Nielsen, Pascal Paillier, Michael Stausholm

Editor:	*Gregory Neven (IBM Research – Zurich)*
Reviewers:	*Ahmad Sabouri (Goethe University Frankfurt)*
	Norbert Goetze (Nokia Solutions and Networks)
Identifier:	*D4.2*
Type:	*Deliverable*
Version:	*1.0*
Date:	*14/08/2014*
Status:	*Final*
Class:	*Public*

Abstract

This deliverable provides an overview of the final reference implementation of the ABC4Trust project.

The cryptographic architecture of the reference implementation has been completely redesigned and re-implemented. This caused only minor changes to the API of the ABCE, showing the versatility of the current ABCE architecture and implementation. New generic services have been implemented allowing easier adoption of the ABCE. Other parts of the ABCE have also seen improvements regarding stability, performance, and usability of the components. Finally, the smartcard source code has also been made available as open source.

The research leading to these results has received funding from the European Community's Seventh Framework Programme (FP7/2007-2013) under grant agreement n° 257782 for the project Attribute-based Credentials for Trust (ABC4Trust) as part of the "ICT Trust and Security Research" theme.

Members of the ABC4TRUST consortium

1.	Alexandra Institute AS	ALX	Denmark
2.	CryptoExperts SAS	CRX	France
3.	Eurodocs AB	EDOC	Sweden
4.	IBM Research – Zurich	IBM	Switzerland
5.	Johann Wolfgang Goethe – Universität Frankfurt	GUF	Germany
6.	Microsoft Belgium NV	MS	Belgium
7.	Miracle A/S	MCL	Denmark
8.	Nokia Solutions and Networks Management International GmbH	NSN	Germany
9.	Computer Technology Institute & Press – DIOPHANTUS	CTI	Greece
10.	Söderhamn Kommun	SK	Sweden
11.	Technische Universität Darmstadt	TUD	Germany
12.	Unabhängiges Landeszentrum für Datenschutz	ULD	Germany

List of Contributors

Chapter	Author(s)
Executive Summary	Janus Dam Nielsen (Alexandra Institute AS)
Introduction	Janus Dam Nielsen (Alexandra Institute AS)
Overview of the Reference Architecture	Janus Dam Nielsen (Alexandra Institute AS)
The ABCE Layer	Hans Guldager Knudsen (Miracle A/S) Kasper Damgård (Alexandra Institute AS) Jonas Jensen (Alexandra Institute AS) Janus Nielsen (Alexandra Institute AS)
New Implementation of the Cryptographic Layer	Patrik Bichsel (IBM Research – Zurich) Robert Enderlein (IBM Research – Zurich)
Smartcard Implementation	Pascal Paillier (CryptoExperts SAS) Thomas Baignères (CryptoExperts SAS)
Demonstration of the Reference Implementation	Michael Stausholm (Alexandra Institute AS)
Performance and Timings	Jonas Jensen (Alexandra Institute AS) Gregory Neven (IBM Research – Zurich)
Conclusion	Janus Dam Nielsen (Alexandra Institute AS)

Executive Summary

This deliverable provides a status update on the reference implementation. Solid progress has been achieved by making the source code of the new cryptographic layer as well as the smartcard code available on the GitHub repository.

The rewritten cryptographic architecture has only slightly impacted the API of the ABCE layer, showing the versatility of the current ABCE architecture and implementation. Other parts of the ABCE have seen improvements as well improving the stability, performance, and usability of the components.

In terms of efficiency, the smartcard turns out to be a bottleneck, taking 80 to 90% of the processing time during presentation. The good news is that the delay vanishes almost completely when using the ABC4Trust stack in a software-only setting, so that one can expect presentation to run within a few seconds.

Table of Contents

Index of Figures

Index of Tables

1. Introduction

This deliverable provides a report on the final ABC4Trust reference implementation. The reference implementation underwent substantial development since the initial reference implementation [GN12].

Most importantly, the underlying cryptographic architecture and implementation has been completely rewritten to make it more modular and more efficient. This change has only slightly impacted the API of the ABCE layer, however, showing the versatility of the current ABCE architecture and implementation.

ABC4Trust has also open-sourced a smartcard implementation referred to as *ABC4Trust Lite* to support the device-binding feature of Privacy-ABC systems. ABC4Trust Lite v1.2 is a dual-interface smartcard application that implements the device-binding versions of both U-Prove and Identity Mixer in a federated way, thereby supporting also other discrete-log-based, device-bound Privacy-ABC systems. The card also features a number of customized functionalities that were required by the pilots, such as counters and encrypted backups. These features can be easily removed or modified to fit other specific needs.

Other substantial changes have been applied in the user client, the browser plugins, and the user interface, which have increased the stability, performance, and usability of the components. We have also measured the performance of the ABCE because the pilot in Söderhamn was rather slow on older hardware deployed in the school. We present a section describing our effort to identify the performance bottlenecks and present some timing numbers.

It is worth noting that the main contribution of the ABC4Trust reference implementation is not the content of this deliverable, but the implementation code and the documentation itself. A new open-source GitHub repository called p2abcengine (for privacy-preserving attribute-based credential engine, https://github.com/p2abcengine/p2abcengine) has been created that makes the Java implementation of the ABCE layer as well as the Javascript implementation of the browser plugin available under an Apache 2.0 license. The Java source code of the new cryptographic library has been made available separately (http://abc4trust.eu/idemix) under a proprietary license that allows for free commercial and non-commercial use. The cryptographic library now implements the algorithms of IBM Identity Mixer as well as of Microsoft U-Prove, meaning that it is no longer necessary to additionally install Microsoft's U-Prove C# Crypto SDK.

This document first briefly recalls the ABC4Trust architecture for a better comprehension of the rest of the document. It then provides more details about the changes to the user client and the cryptographic engine, and gives an overview of the smartcard code. In the final section, we present some performance results.

2. Overview of the ABC4Trust Architecture

The ABC4Trust architecture is the foundation of the reference implementation. In this section, we give a brief overview of the ABC4Trust architecture, to help understand the different components. We also describe a technical detail discovered since deliverable D4.1 [GN12] regarding the practical usage of U-Prove.

The reference implementation faithfully implements and realises the architecture, protocol, and API for Privacy-ABCs as defined in Deliverable D2.2 [BCD+14]. The overall entities of the ABC4Trust architecture are the user, the issuer, the verifier (sometimes also referred to as the relying party), the revocation authority, and the inspector. Their interaction is defined in a number of protocols explained in D2.2 [BCD+14]. Figure 1 depicts the protocol and software architecture for presentation of a credential. The user is on the left side of the figure, the relying party on the right, the revocation authority is depicted at the bottom.

A user initiates the protocol by requesting some resource from a relying party. For example, he may request access to some information on a website. The relying party replies with a presentation policy that the user must satisfy. The user invokes the ABCE layer to devise a proof for the satisfaction of the policy. The ABCE layer manages the user's credentials and pseudonyms. These may be stored locally or on a smartcard, in which case they are fetched through the external device data interface. The ABCE layer matches the incoming presentation policy against the users' credentials and pseudonyms to infer the different ways in which the user can satisfy the presentation policy. This may imply fetching the latest revocation information from the revocation authorities associated with the user's credentials. The user is then offered a choice between the different possibilities through the identity selection user interface. When the user indicates his choice, the ABCE layer instructs the underlying cryptographic layer, called the crypto engine, to create the cryptographic evidence for the selected presentation token. If the credentials involved are bound to a smartcard, then the crypto engine will invoke that smartcard through the external device crypto interface.

The user then sends the presentation token to the relying party. The ABCE layer checks that the received presentation token indeed satisfies the presentation policy, while the cryptographic layer verifies the cryptographic evidence of the token. If all checks succeed, the relying party allows the user access to the requested resource.

The protocol for issuance is very similar to the protocol for presentation, as the user might need to present a number of credentials or attributes to get a new credential issued.

The protocol for inspection is very simple; the issuer or verifier sends the presentation token to the inspector, who will decide to decrypt based on the inspections grounds. The decrypted attributes may be sent back to the requestor, or forwarded to the instance that is to be informed about the user's identity (e.g., police, school board, etc.).

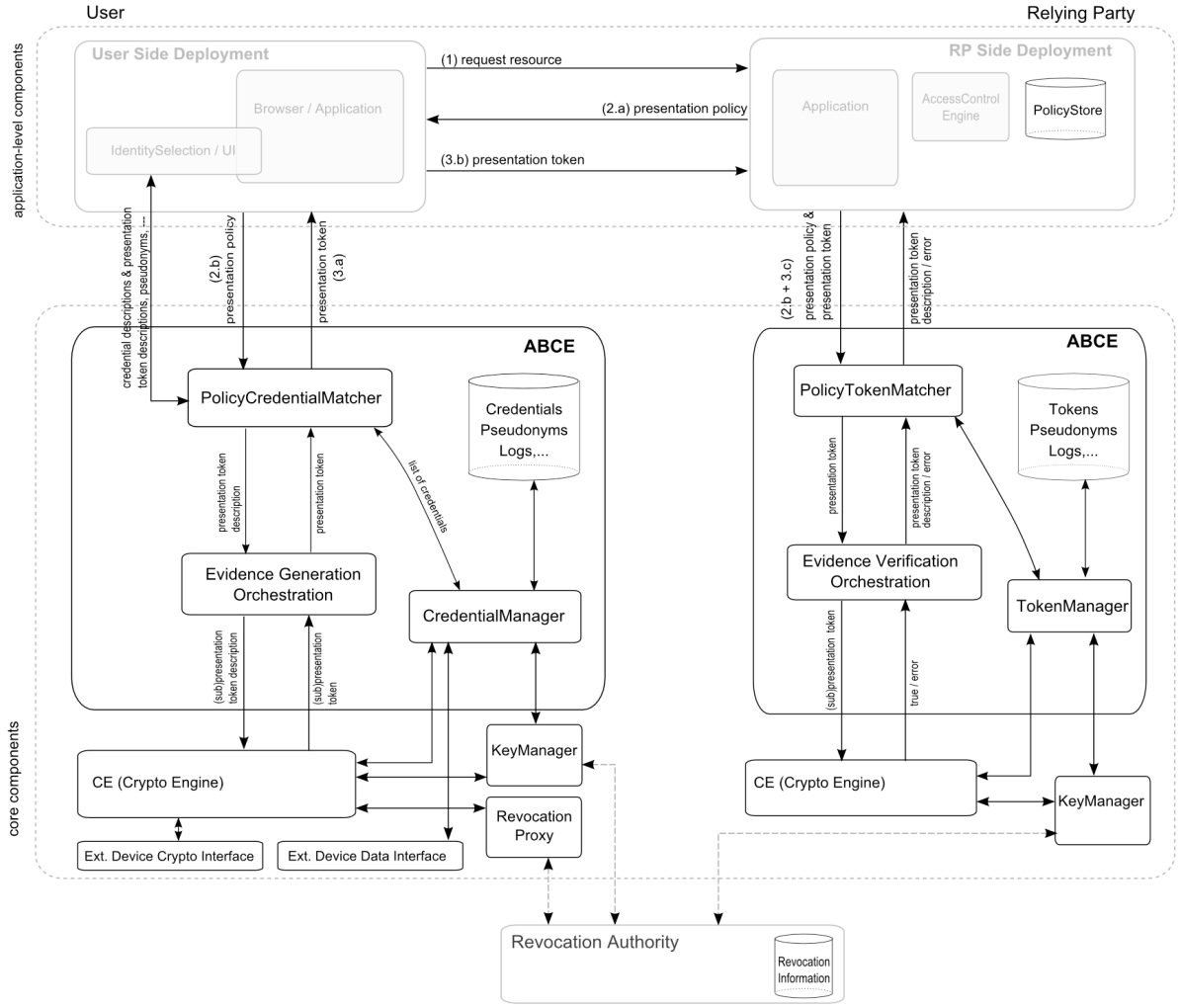

Figure 1. Presentation of a credential

Due to the nature of U-Prove tokens, they can only be used once in certain types of presentations (if the user needs to be unlinkable between presentations, e.g. when using fresh pseudonyms). This implies that the number of U-Prove tokens that the user obtained at issuance will decrease over time, eventually to depletion. In order to continue usage of the Privacy-ABC technology, the user must obtain new U-Prove tokens. New tokens are obtained using a protocol that we call *reissuance*, as the user must contact the issuer to obtain new U-Prove tokens.

In the ABC4Trust project, reissuance was in particular required for the Söderhamn pilot, where students are expected to use U-Prove tokens quite often. The Patras pilot only involved at most two unlinkable presentations for each credential (one for filling out the evaluation form, one for participating in the tombola), so reissuance was not needed there.

The smartcard stores, for each credential, which URL the user should visit to perform reissuance. When there are no more U-Prove tokens available, the user service automatically contacts those URLs and initiates the reissuance protocol. The protocol is identical to the issuance, apart from the issuer using the same value for the revocation handle that was also used in the original issuance. This way, the user can keep his old credential, but use the new U-Prove tokens.

3. The ABCE Layer

In this section we report on the current status and development of the reference implementation. The implementation is close to feature-complete with respect to the functionality described in D2.2 [BCD+14], although there are still a number options for improvements, future development, and new applications. The level of maturity and completeness was demonstrated by the fact that changing the underlying cryptographic architecture during the last year did not cause any substantial changes to the interface of the ABCE. This is a remarkable result, which shows the versatility of the current ABCE architecture and implementation.

One of the objectives of this project was to deliver an open reference implementation, which is available to the interested stakeholders. The ABCE layer of the reference implementation has been made available as part of the privacy-preserving attribute-based credential engine GitHub repository (https://github.com/p2abcengine/p2abcengine), thus making it available to a broad audience. In addition to the ABCE layer, the GitHub repository contains also the Javascript implementation of the browser plugin.

Apart from the source code, the release includes documentation for how to setup and integrate the ABCE. The wiki associated to the GitHub repository contains a summary of the concepts and features of the ABCE (https://github.com/p2abcengine/p2abcengine/wiki/Concepts-and-Features), a high-level overview of the architecture (https://github.com/p2abcengine/p2abcengine/wiki/Architecture), instructions to integrate Privacy-ABCs into an existing application using helper classes (https://github.com/p2abcengine/p2abcengine/wiki/Integrating%20the%20ABC-Engine), as well as sample code in the form of Unit tests (https://github.com/p2abcengine/p2abcengine/tree/master/Code/core-abce/abce-services/src/main/java/eu/abc4trust/services).

The first iteration (based on the old cryptographic architecture) of the core ABCE that was made available on Github was deployed in the first and second round of the Söderhamn trial. The main differences with the current GitHub offering are the configuration of the ABCE and the addition of the generic services. The Patras pilot used the new cryptographic architecture.

The publicly available ABCE consists of a number of core components and a user interface needed to implement a Privacy-ABC system. The components deal with issuing, verifying, inspecting, and revoking privacy-preserving attribute-based credentials, as well as handling the required user interaction.

The reference implementation on Github is released under the Apache 2.0 open source license. This license is a very permissive open source license, which enables a broad range of companies to use the reference implementation, and develop many different kinds of applications. The cryptographic engine underlying the ABCE must be obtained separately (http://abc4trust.eu/idemix).

In the following sections, we will report on various aspects of the reference implementation that have not been covered in D4.1, or have received substantial attention after the submission of D4.1. This includes the user client, browser plugins, and user interface, which have seen substantial development.

We also present two sections on implementation details, which are based on choices we had to make during the development. One section is on limitations imposed by our choices, the other is on the choice of how to store various artefacts (keys, credentials, parameter, etc.) and their alternatives. We also report on the deployment of the ABCE as web services where we describe the underlying abstractions introduced to make the implementation of the web-services easier. Furthermore, we describe the web-services and their deployment and usage. Finally, we discuss an aspect of quality assurance.

3.1 About the Reference Implementation

The main purpose of the reference implementation is to provide a common platform that exposes a unified architecture for selected Privacy-ABC systems and to enable the deployment of the reference implementation in pilots involving actual end-users. Furthermore, the reference implementation is a research vehicle for understanding the interoperability issues of the selected ABC systems.

It is not a goal of the reference implementation per se to be a highly performance-tuned implementation. The reference implementation is not dedicated to a particular usage scenario, but implements a generic Privacy-ABC system, and as such is very versatile. The reference implementation can easily be deployed and integrated as discussed in section 3.6.2, but it will require some tailoring to fit the specific needs of a particular application. The main modifications points will be the user interface, smartcard application, storage (of keys, credentials, parameters etc.), and perhaps performance. An example is the identity selection user interface, for which we provide a generic implementation that works for any presentation policy, but that can probably be simplified in concrete application cases.

3.2 User client, browsers, GUIs, and user service

3.2.1 User Client

The User Client contains the part of the ABCE that runs on the client side. It takes care of the client side of issuance, and makes sure that the resulting credential and other information is stored at the correct places, be it on the smartcard or disk. It also generates proofs for presentation when asked and is responsible for contacting the smartcard if this is needed during the proof generation. It can also present various information such as a list of all available credentials and the status of those, and toggling timings on/off in the ABCE. All of this functionality is available through the web service API calls, but a web service application making use of the ABC4Trust framework has to go through the browser plugins, where all of the communication goes through (see Section 3.2.3). Communicating directly with the User Client from a website should be avoided due to cross-site scripting. The User Client itself can be wrapped inside the User Service (Section 3.2.2) if one would like to do so.

3.2.2 User Service

The user service is a Windows service to make it easier for the users to use our software. It starts up automatically and makes the interaction between the user service and the browser plugin in such a way that it is transparent for the user. The user service can work also with a software smartcard, and is easy to enable by just including a single file in the corresponding folder.

3.2.3 Browser plugins

The browser plugins are responsible for the communication between the website that the user is visiting and the user service. The browser plugins simply relay messages between the website and the local ABCE stack. They are also responsible for showing the Identity selector GUI, which we describe in Section 3.2.4. Additionally there is a menu with the following features:

- **Manage credentials:** Launches a window where the user can browse his credentials to view their contents and their revocation status. If enabled, credentials can also be deleted manually.
- **Check revocation status (optional):** This function checks, for all credentials, whether they are revoked, and deletes them if that is the case. This creates more space on the card and makes sure that no credentials are deleted which could still be of use. This can be considered as a pilot-specific feature for the Söderhamn pilot.

- **Check Attendance Data (optional):** Returns the current attendance count as well as the threshold for entering the course evaluation. It is a pilot-specific feature for the Patras pilot.
- **Backup smartcard:** This function enables the user to take a backup of his or her smartcard. It stores encrypted content of the smartcard such as the device secret using a key known only to this type of smartcard combined with the user's password that he is prompted to enter. The information that requires the user's pin code, such as the blob store, is encrypted using the user's password.
- **Restore smartcard:** Given that a backup file is present on the computer, this function will, given a password, try to restore the smartcard to the state the card had when backup was performed. It can only be done if the smartcard has the same device ID as the card had when backup was performed. This ID is set at initialization time, which should prevent any easy duplicate cards from showing up.
- **Change pin:** This function changes the PIN of the card given that the original PIN was correct.
- **Unlock card:** This function takes the PUK code as input as well as the new PIN, and can unlock a locked card. A card is locked if the PIN is entered incorrectly 3 times in a row.

These features can be seen in Figure 2, which shows the ABC4Trust menu items. In Figure 3 we see the credential management window.

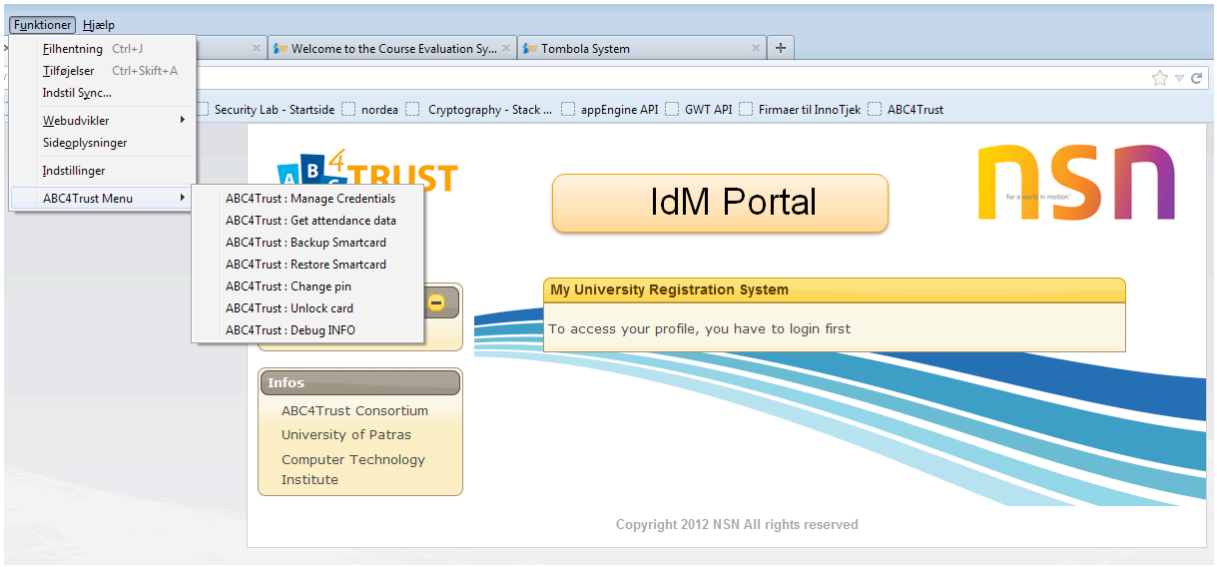

Figure 2. The Firefox plugin menu.

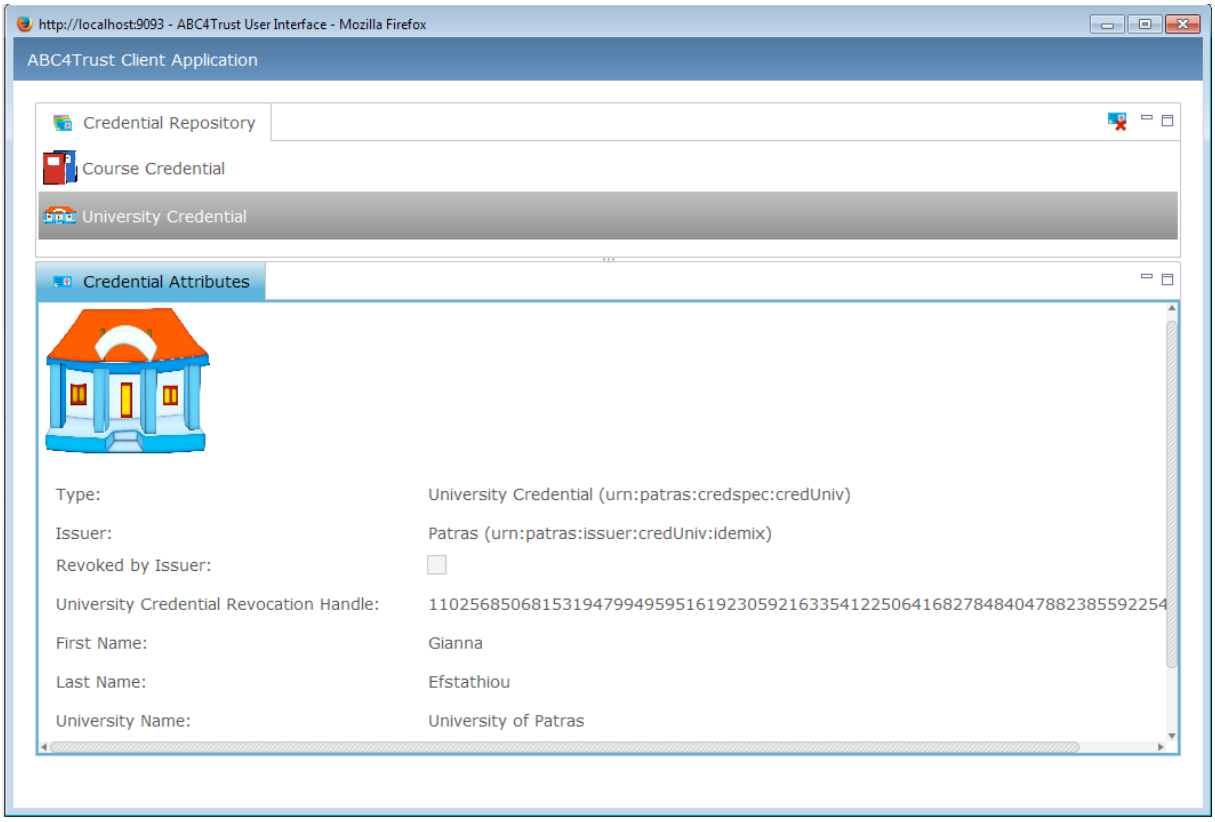

Figure 3. Credential overview

3.2.4 Identity Selection GUI

In order to get the user to choose which kind of identity to reveal to a verifier, we built the Identity Selection Graphical User Interface (GUI). This is the only place where the user gets informed of what is revealed. A normal website could claim whatever it wants, but the GUI runs locally based on information from the user-service and thus the user's smartcard. It is needed since there might be several ways to fulfil a certain policy. Maybe a website would have to know that you either are a member of the website or that you are above 18 years old. If the user can prove both, he will be presented with the choices. The Identity Selector will only show possibilities that the user can actually prove something about. The Identity Selector also shows the inspection grounds if such are present. An example of the Identity Selector can be seen in Figure 4.

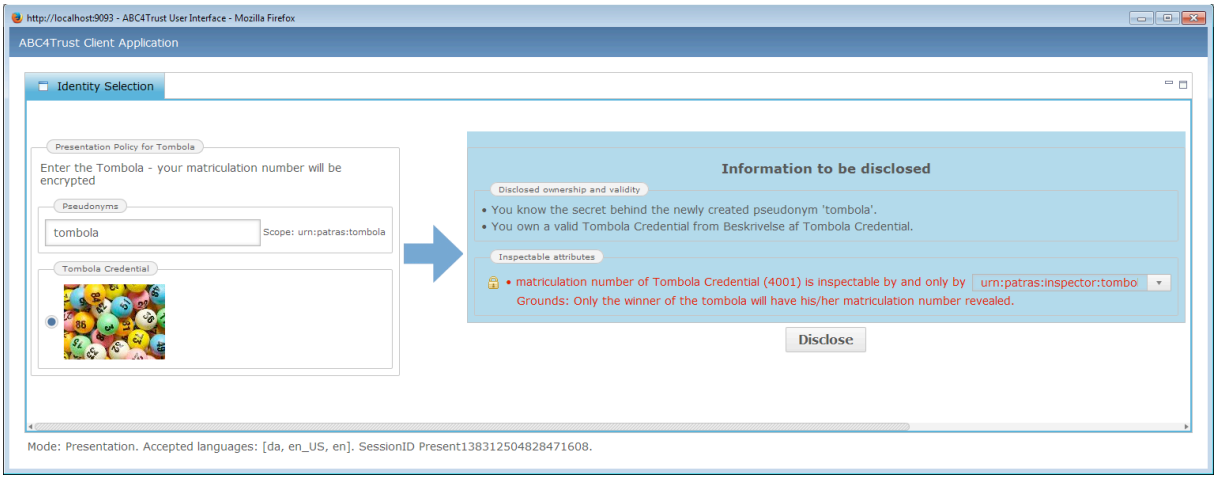

Figure 4. Identity selector

3.2.5 Communication between the Browser Plug-in and Issuer, Verifier, and Revocation Authority

The communication between the different ABCE engines is done using REST-like web services. This means that the ABC4Trust XML messages are retrieved and sent using the HTTP protocol. In particular we use the GET and POST methods. The web-service endpoint for contacting Issuer and Verifier is set up in Javascript when configuring the browser plugin. The web-service endpoint towards the Revocation Authority is set up in the Revocation Authority Parameters. Implementers of ABCE services must implement the services specified below.

3.2.5.1 Issuer Web Methods

The Issuer needs to specify two resources (URLs) for the browser plugin/user client. The first is for retrieving the Issuance Policy. The second is where to post Issuance Step messages. Web session information can be encoded into the URLs.

Table 1. Retrieve Issuance Policy

HTTP Method	Payload and arguments	Return type	Return codes
GET	None	IssuanceMessage (containing Issuance Policy)	HTTP 200 for a positive invocation. HTTP 4XX/5XX when rejecting the request.

Table 2. Send Issuance Step Message

HTTP Method	Payload and arguments	Return type	Return codes
POST	IssuanceMessage	IssuanceMessage	HTTP 200 for a positive invocation. HTTP 4XX/5XX when rejecting the request.

If a request is rejected the Issuers HTTP response code will be available to the Javascript in the Issuers Web Application.

3.2.5.2 Verifier Web Methods

The Verifier needs to specify two URLs for the browser plug-in/user client. The first is for retrieving the PresentationPolicy. The second is where to post the PresentationToken messages. Web session information can be encoded into the URLs.

Table 3. Retrieve presentation policy

HTTP Method	Payload and arguments	Return type	Return codes
GET	None	PresentationPolicyAlternatives	HTTP 200 for a positive invocation. HTTP 4XX/5XX when rejecting the request.

Table 4. Verify token against policy

HTTP Method	Payload and arguments	Return type	Return codes
POST	PresentationToken	IssuanceMessage	HTTP 200 for a positive invocation. HTTP 4XX/5XX when rejecting the request.

If a request is rejected, the Verifiers HTTP response code will be available to the Javascript in the Verifiers Web Application.

3.2.5.3 Revocation Authority Web Methods

The URLs for the Revocation Authority is handed over to the user-clients in the Revocation Authority Parameters. The 'private' URL/WebMethod for doing actual revocation is not specified in the revocation authority parameters, and must be agreed upon between the Revocation Authority and the organisation initiating the revocations.

Table 5. Generate non-revocation evidence

HTTP Method	Payload and arguments	Return type	Return codes
POST	RevocationParameterUID specified as argument, AttributeList as payload	NonRevocationEvidence	HTTP 200 for a positive invocation. HTTP 4XX/5XX when rejecting the request.

Table 6. Generate non-revocation evidence.

HTTP Method	Payload and arguments	Return type	Return codes
POST	RevocationParameterUID specified as argument, epoch as payload	NonRevocationEvidence	HTTP 200 for a positive invocation. HTTP 4XX/5XX when rejecting the request.

Table 7. Get revocation information

HTTP Method	Payload and arguments	Return type	Return codes
POST	RevocationParameterUID specified as argument	RevocationInformation	HTTP 200 for a positive invocation. HTTP 4XX/5XX when rejecting the request.

3.3 Implementation of storage and alternatives

It is not obvious where and how one should store the different artefacts. This section contains thoughts about the different possibilities and what we ended up choosing.

First of all, there are different entities that store different things: Issuer, revocation authority, verifier, inspector and the user. We will here go through each of them. All entities have one thing in common, namely the notion of a *key storage*. This is just a simple file, and holds not only keys, but also other resources that are needed at runtime. The reason for having such storage is that it makes the internal code simpler when it can just rely on the position in a single file rather than knowing the filenames for the resources it needs. An initial setup is then needed to import these files into the key storage.

3.3.1 Issuer

The flow starts with the issuer, as he is the one who has to generate the issuer parameters. These have to be stored somewhere. The parameters consist of two parts – a private and a public part. The private part of course has to be known to the issuer only. These we store on the disk of the issuer serialized down as java objects. This takes up slightly more space than if we would have serialized as bytes, but it is easier and less error-prone. Also, since the sizes we work with rarely get to even MB sizes, this is not an issue.

The same technique is used for the public parts, but here there is no requirement that they should be kept secret, and could thus be located in e.g. the cloud. We chose the less complicated method of storing to disk and then send the files around to those other entities who need them, as this was less time-consuming, and the gain of a cloud solution is limited.

The last resources that are also generated and stored by the issuer are the system parameters and credential specifications, which can be seen as another public artefact. These are both stored as a java object and as XML to make it possible to easily read the content of the files.

3.3.2 Revocation Authority

The revocation authority stores only the revocation authority parameters which are stored as a Java object for ease of use. This is a public artefact, and is thus shipped around to the ones who need it. We could, as with all public artefacts, have stored it in the cloud for ease of access, but the gain is again too small to spend the effort to do so.

3.3.3 Verifier

The verifier does not in principle need to store anything, besides the issuer and system parameters, but by default it stores the presentation tokens it verified. This is done as java objects serialized into a file. The reason we want this is to later inspect a token if needed. In a system without inspection, you would not want to store the presentation tokens as they are not really of any use later.

3.3.4 Inspector

The inspector has, like the issuer, a private and a public part of a key. The private part should be stored on a permanent storage device where it is secure. The inspector is the only one who can reveal the real identity of users (if inspection feature was enabled for the presentation), meaning that his private key is the most vital to the system. Thus, we decided to use a smartcard for the storage of the private key. This is secured via pin-code and not connected to the Internet, so it requires physical access and knowledge of the PIN code. We could also have stored this on a computer, but the best protection would then have been to remove the hard drive and store it in a safe. This is much more cumbersome, and a smartcard can be backed up easily as well by just making more copies.

The public part of the key is once again just serialized as a Java object and shipped to the relevant entities. This could, since it is public, also be stored in the cloud, but again because of the small gain of doing so, we chose the file-based solution.

3.3.5 User

The credentials, pseudonyms and other information needed by the applications, are stored on the smartcard.

For the user, there is a difference in what is stored where based on the technology we use. If U-Prove is used, the issuance protocol results in 1 or more U-Prove tokens. These need to be stored somewhere to be used at later presentations. We chose two different approaches for the two pilots. For Patras, since they only need a single token, we store the token on the smartcard, as there is plenty of space. For Söderhamn, they are required to be able to use a lot of tokens, which would take up too much space on the smartcard. Thus, we have to store those tokens on another permanent storage device. The obvious choice is the disk of the computer. This means, though, that a user would have to be at the same computer when logging in as when he got his credential issued. If the user is at a different computer, he would have to use the re-issuance protocol described in Section 2. However, since the U-Prove tokens are linked to the user's smartcard, meaning they could in principle be put into the cloud, an adversary cannot use the tokens. This would have been the better solution and we would investigate it later, but the time limitations forced us to use the disk solution.

3.4 Limitations

In the development of the reference implementation, we have come across a number of situations where there either is no good solution, or the good solution would be very time consuming without add much value to the project. In this section we intend to describe these situations, how they have been dealt with, and what the consequences are.

3.4.1 Multiple Smartcards Support

When implementing the support for smartcards in the user client, we were faced with the question of how to handle the case where a user attaches two or more smartcards to the same computer. This is not a very likely case, but might happen if two users share the same computer e.g. at home and leave the cards lying on the table very close to the smartcard reader. Another scenario is if a curious user attaches two smartcards. We estimate that these cases are very rare. The user must enter a pin before the smartcard can be used. The user client stores the pin temporarily in memory and must present the pin to the smartcard for most operations. This leaves the possibility of a user experiencing a malfunction, e.g. a presentation, which should have verified is rejected, or inadvertently gets issued a credential on the work smartcard to be very close to not existing.

If multiple smartcards are attached to the computer, then the ABCE will either produce an error or pick a random, depending on the parts of the implementation. An alternative would be to allow the user to choose a specific smartcard. This however, is very much effort compared to the size of the almost non-existing problem.

3.4.2 Multiple Secrets Support

In line with the previous comments on multiple smartcards, we also do not support multiple secrets. A secret can be bound to a device, e.g. a smartcard, or not e.g. stored on the computer. However, it is very unlikely if a user has multiple secrets stored on the same computer. And as such we have chosen not to allow more than one key in the ABCE.

An alternative would be to allow the user to choose a specific secret for each operation in the case of multiple secrets being available. However, this is much effort with only very little value to the project and likely applications.

3.5 Description of “Helpers”

A number of so-called “helpers” were developed to facilitate the development of integration tests and to simplify the implementation of web-services exposing the issuer, verifier, user, inspector, and revocation authority entities.

The “helpers” have been used to verify that each ABCE would run separately in its own process, being able to load and store resources so that it can survive shut down and restart and communicate the ABC4Trust XML messages over a network.

Basically the helpers contain all the boilerplate code, which an implementer would have to write to configure an ABCE. When configuring a “helper” one specifies a storage directory on the file system and where to export resources generated for other ABCEs (if any) and which external resources to import.

We have created five integration test projects, one for each ABCE role. In the integration test projects the ABCE modulesare then individually embedded in a REST service and the integration test runner is setup to run a couple of test scenarios. Common for all of the tests are to setup an Issuer, User and Verifier. Thus we reuse the “helpers” through the tests.

3.5.1 Issuance Helper

The “Issuance Helper“ is used to kick start an ABCE infrastructure. It can either generate or use existing system parameters. When setting it up the helper is told which credential specifications are supported and which issuance policy to use when issuing the credential. If the credential specification is specified to be revocable the reference to the revocation authority must also be specified. Based on this it will then automatically generate an issuer parameter for the credential specification.

After initialization system and issuance parameters are exported for the other ABCEs to pick up.

When starting an issuance the "helper" can perform basic validation of the attributes values, which are going into the credential. It will also check if the issuance policy is referring to credentials, which are revocable. If this is the case the revocation information will be retrieved from the Revocation Authority and added to the policy.

3.5.2 Revocation Helper

When setting up the "Revocation Helper", the UID of the Revocation Authority is specified together with the URLs where the Revocation Authority web-service will respond to the different queries.

After initialization of the revocation authority, parameters are exported for other ABCEs to pick up.

3.5.3 Inspection Helper

Configuring the "Inspection Helper", one specifies the UID of the inspector public key and the credential specifications for the credentials, which are going to be inspected.

After initialization, an inspector public key is exported for the other ABCEs.

3.5.4 Verification Helper

The "Verification Helper" is setup to use resources from Issuer, Revocation Authority, and Inspector.

Before sending a presentation policy to a user the helper checks if the presentation policy is referring to credentials that are revocable.

If this is the case, the revocation information will be retrieved from the revocation authority and added to the policy.

3.5.5 User Helper

The "User Helper" is setup to use resources from Issuer, Revocation Authority and Inspector.

3.6 Description of generic services

The reference architecture described in D2.2 [BCD+14] defines a web-service interface to the ABCE. A web-service interface allows the ABCE to be deployed using standard techniques and makes integration less time consuming by exposing a rest-based interface. In this section we describe how to build, setup, and integrate the ABCE issuer, user, verifier, inspector, and revocation web-services.

3.6.1 Building the ABCE Web-services

The ABCE web-services are implemented using Java and Jersey; the JAX-RS reference implementation is used as the foundation for exposing the service interface. To build the ABCE web services, one must first set up the build environment for the core ABCE components. Instructions on how to do so can be found on the GitHub page (https://github.com/p2abcengine/p2abcengine/wiki/How-to-Build-the-ABC-Engine).

The services can be compiled using a standard Java tool chain based on maven. Each service can be compiled either as a self-contained executable or a war-file ready for deployment in a standard application server/servlet container, e.g., Tomcat. The self-contained executable is comprised of the service implementation bundled with the Jetty servlet container, and allows for easier installation e.g. for demonstration purposes.

The source code for the services is located in following path of the GitHub project repository:

```
Code/core-abce/abce-services
```

All commands are relative to this path. First, a clean Maven 3 install must be run in the parent directory.

```
mvn clean install -DskipTests
```

This will download dependencies and compile the ABCE for inclusion into the services.

The self-contained executable is created by instantiating the following command template with the proper value for "{service-name}" from the set: issuance, verification, revocation, user, and inspection.

```
mvn -P selfcontained-{service-name}-service install -DskipTests
```

The war-files suitable for deployment in a servlet container are compiled by instantiating the following command template with the proper value for "{service-name}" from the set: issuance, verification, revocation, user, and inspection.

```
mvn -P {service-name}-service install -DskipTests
```

A bash script found in the ABCE-services directory has been created to generate both kinds of artefacts in an automated and reliable manner. The script is executed using the following command (remember to check permissions of the script if it fails) from the ABCE-services directory:

```
compileServices.sh
```

The artefacts can be found in the abce-services/target directory.

3.6.2 Setup and Running the ABCE Web-services

The self-contained executable is executed as a standard Java jar file. The port number is an optional argument with the default value of 9500. The following command template can be used to make it run on a port defined by the value of "{port-number}":

```
java -jar selfcontained-{service-name}-service.war {port-number}
```

Each service will create storage directories as direct subfolders to the directory in which they are executed.

The war-file can be deployed in the standard ways defined for the given servlet container. They will be available on the port number as configured in the servlet container.

3.6.3 Running the ABCE Web-services in Debug Mode

The self-contained web-services can easily be made available for a remote debugger. This is highly useful for identifying bugs during development of the ABCE web-services. The following command template can be instantiated to enable a standard Java debugger to attach to the web-service process on port 4000:

```
  java -Xdebug -Xrunjdwp:transport=dt_socket,address=4000,server=y,suspend=n -jar
target/selfcontained-{service-name}-service.war {port-number}
```

3.6.4 Using the ABCE Web-services

The ABCE web-services expose a REST over HTTP interface defined in D2.2, which is a superset of the interface defined in D4.1. The main difference is the addition of endpoints for storing, e.g., system parameters and credential specifications on the relevant web-services.

We will give a brief introduction to the web-services interface by presenting excerpts of a tutorial that we developed to demonstrate the usage of the web-services. Since the complete tutorial is quite substantial, only a few minor parts are documented here. The full tutorial is available as part of the reference implementation source code, where it can be found in the abce-services directory. A bash script was made to automatically execute the tutorial given a setup of the web-services. We refer to D2.2 for an in-depth description of the interfaces.

The tutorial expects you to be familiar with Privacy-ABC technologies and the architecture of the ABCE as described in D2.2. The tutorial will therefore mainly focus on how the web services are used and not on the Privacy-ABC components, i.e. we assume that meaningful artefacts such as credential specifications and issuance / presentation policies already exist. The tutorial is based on a scenario where a soccer team sells tickets for their home matches using an external ticket handling company (e.g. like TicketMaster.com), which will issue tickets on behalf of the soccer team. The soccer team employees will then verify the tickets at the soccer arena on the day of the match to make sure that only paying customers are allowed in. Furthermore, to illustrate use of inspection and revocation, we make the tickets revocable and allow the soccer team to provide special treatment to VIP customers, who will enter in a raffle if they show up (requiring inspection).

The scenario can naturally be extended to cover many use cases where an organisation, e.g., a theatre or a museum, sells tickets to one or more events.

The full scenario consists of the following steps:

1. Define credential specification, issuance policies, and presentation policies
2. Setup system parameters
3. Generate inspector keys
4. Setup revocation authority parameters
5. Setup issuance parameters
6. Issue ticket (credential)
7. Present ticket (generates a presentation token)
8. Inspect VIP number (which is contained as inspectable attribute in the presentation token)
9. Revoke ticket
10. Retry presentation, which will fail because the ticket has been revoked

Remark steps 3, 8, 9, and 10 are left for future work.

The first step is to define credential specifications. The XML below shows the credential specification for a VIP ticket, which we use in this tutorial.

```
<abc:CredentialSpecification xmlns:abc="http://abc4trust.eu/wp2/abcschemav1.0" Version="Version 1.0"
KeyBinding="true" Revocable="true" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://abc4trust.eu/wp2/abcschemav1.0
abc4trust-xml/src/main/resources/xsd/schema.xsd">
    <abc:SpecificationUID>http://MyFavoriteSoccerTeam/tickets/vip</abc:SpecificationUID>
    <abc:FriendlyCredentialName lang="en">VIP ticket to soccer match</abc:FriendlyCredentialName>
<abc:DefaultImageReference>http://MyFavoriteSoccerTeam/tickets/vip/img</abc:DefaultImageReference>
    <abc:AttributeDescriptions MaxLength="256">
        <abc:AttributeDescription Type="http://abc4trust.eu/wp2/abcschemav1.0/revocationhandle"
        DataType="xs:integer" Encoding="urn:abc4trust:1.0:encoding:integer:unsigned"/>
            <abc:FriendlyAttributeName lang="en">Revocation ahndle</abc:FriendlyAttributeName>
        </abc:AttributeDescription>
        <abc:AttributeDescription Type="FirstName" DataType="xs:string" Encoding=
        "urn:abc4trust:1.0:encoding:string:sha-256">
            <abc:FriendlyAttributeName lang="en">Name</abc:FriendlyAttributeName>
        </abc:AttributeDescription>
        <abc:AttributeDescription Type="LastName" DataType="xs:string" Encoding=
        "urn:abc4trust:1.0:encoding:string:sha-256">
            <abc:FriendlyAttributeName lang="en">Lastname</abc:FriendlyAttributeName>
        </abc:AttributeDescription>
        <abc:AttributeDescription Type="Birthday" DataType="xs:date" Encoding=
        "urn:abc4trust:1.0:encoding:date:unix:signed">
            <abc:FriendlyAttributeName lang="en">Birthday</abc:FriendlyAttributeName>
        </abc:AttributeDescription>
        <abc:AttributeDescription Type="Matchday" DataType="xs:date" Encoding=
        "urn:abc4trust:1.0:encoding:date:unix:signed">
            <abc:FriendlyAttributeName lang="en">Matchday</abc:FriendlyAttributeName>
        </abc:AttributeDescription>
        <abc:AttributeDescription Type="MemberNumber" DataType="xs:integer" Encoding=
        "urn:abc4trust:1.0:encoding:integer:unsigned">
            <abc:FriendlyAttributeName lang="en">VIP member id</abc:FriendlyAttributeName>
        </abc:AttributeDescription>
    </abc:AttributeDescriptions>
</abc:CredentialSpecification>
```

Figure 5. Credential specification for soccer tickets.

The next step after we have defined the credential specification is to create the system parameters. An issuer, in our case the ticketing company, generates the system parameters. The issuer executes a HTTP request against the issuer web-service. We show it here using curl, which is a program for executing HTTP requests.

```
 curl -X POST --header 'Content-Type: text/xml'
'http://localhost:9100/issuer/setupSystemParameters/?securityLevel=80&cryptoMechani
sm=urn:abc4trust:1.0:algorithm:idemix' > systemparameters.xml
```

In this case curl is instructed to make a HTTP POST request against an issuance service running on port 9100 at localhost. We target the setupSystemParameters resource and provide the security level and crypto mechanism as arguments. The security level ultimately dictates the level of keys to be used in the deployment, which uses the system parameters, e.g., 2048 bit. The crypto mechanism can be one of Idemix or U-Prove. The resulting system parameters are written to a file called systemparameters.xml. This process, depending on the hardware, may take more than 15 seconds.

Next the system parameters must be distributed among the other parties (users, verifier(s), revocation authority/ies, and inspector(s)).

As an example, we here show the HTTP request for storing the system parameters at the revocation authority:

```
 curl -X POST --header 'Content-Type: text/xml' -d @systemparameters.xml
'http://localhost:9500/revocation/storeSystemParameters/' >
storeSystemParametersResponceAtRevocationAutority.xml
```

The revocation authority needs to generate revocation authority parameters before the issuer can generate issuer parameters.

```
 curl -X POST --header 'Content-Type: text/xml' -d @tutorial-
resources/revocationReferences.xml
'http://localhost:9500/revocation/setupRevocationAuthorityParameters?keyLength=1024
&cryptoMechanism=urn:abc4trust:1.0:algorithm:idemix&uid=http%3A%2F%2Fticketcompany%
2Frevocation' > revocationAuthorityParameters.xml
```

Here the key length[1], crypto mechanism, and revocation authority parameters UID are given as arguments. The UID is URL encoded.

The revocation authority parameters and credential specification must be distributed among the issuer, users, and verifier. The commands are similar to the one for storing system parameters.

The rest of the tutorial is available online at the Github repository[2].

3.6.5 Deployment and Integration of the services

As mentioned above the issuance, verification, revocation authority, and inspector services can be deployed either standalone or in a servlet container. This allows for the ABCE to be deployed and maintained in the same way as other Java servlet based infrastructure. The ABCE web-services are not meant to be facing a network e.g. a corporate network or the Internet. A layer of consisting of at least a firewall is required, and often also a layer containing some level of business logic. An example is the issuer, which provides a wide range of resources. Some of them like *issuanceStep* should be available to users and others like *setupSystemParameters* or *setupIssuerParameters* should not be widely exposed. Also *initIssuanceProtocol* should not be directly exposed to the network, but rather only be exposed to some business application, for example by letting the ABCE service accept only connections from a thin web application that faces the public network and that can filter requests. which can do the initial validation of users, and lookup their attributes, which should be provided to the issuance service as input to the *initIssuanceProtocol*.

The web-services do not currently provide any authentication mechanisms. This means that anybody who has network access to the services can send commands to the services. This does not threat the secret keys stored at each services. However, service interrupts can be easily be accomplished by e.g. requesting new revocation authority parameters with the same UID as the current parameters. This will overwrite the parameters at the revocation authority and results in requests being rejected or responses being malformed. Therefore we recommend that the ABCE web-services are run behind a firewall and that only the public resources are accessible from outside the firewall.

[1] We are working on harmonizing the interface so the revocation authority should in the future be made to take the security level as input instead of the key length.

[2] See https://github.com/p2abcengine/p2abcengine/

The user services also support the same flexible deployment options. The user service would most likely be deployed as a self-contained executable, which can be installed on users' machines. The project has created a Windows installer which downloads a precursor for user web-service as part of the installation process and installs the user web-service as Windows services. And thus makes it available for the browser plugins which also have been developed and which provide the graphical user interface to the supported user actions.

3.6.6 Usage of the generic ABCE Web-services in the pilots

The generic ABCE web-services have only recently been implemented, which means that most of them have not been ready for the pilots. A set of pilot specific services have been developed in time for the pilots and used in the initial pilot round. Since both the pilot specific and the generic web-services share a lot of functionality, the foundation of the pilot specific services (the so-called service helpers) has been reused as the foundation of the generic ABCE web-services. The main difference between the pilot specific services and the ABCE web-services is the interface provided, in particular the addition of the ways to store various parameters.

By the time of the second round of the Patras pilot, we had implemented some of the generic web-services. Since the pilot requires a revocation authority, we had previously used a pilot specific revocation authentication web-service. For the second round of the Patras pilot however, we changed to the new generic service to utilize the reference implementation and to verify that the generic services would work in practice.

4 New Implementation of the Cryptographic Layer

The reason for providing a new implementation of the cryptographic layer of the Privacy-ABC system was mainly the missing modularity of the former library. The latter did not allow for expanding its functionality with further protocols. That is, the addition of the Brands signature scheme [Bra99] used by the U-Prove library was not possible. This posed challenges when it comes to the integration of different cryptographic algorithms into the same protocol run and thereby limits the overall technologies and concepts from being adopted.

The new Identity Mixer library implementation solves this problem by providing building blocks for all abstract functionalities. As an example, the signature building block abstracts from the specifics of the algorithms such as Brands or Camenisch-Lysyanskaya [CL01] signatures and only exposes their common functionality, which builds the basis of privacy attribute-based credentials.

In the design of the cryptographic layer, to which we also refer as the crypto engine, the external interfaces that the ABC system specifies were taken into account. Mostly, the current implementation follows these specifications and re-uses many data formats as specified in [BCD+14] but it diverges where appropriate as the detailed specification of the cryptographic layer [BCE13] show. The following description largely follows their overview of the architecture.

The crypto engine can be divided into (1) a largely crypto-agnostic part that assembles data and prepares it and (2) a collection of abstract building blocks and their concrete implementations. The first part can be further divided into two layers:

- The first layer consists of the proof orchestration (singleton) object, the issuance orchestration object, the building block factory, and in general all singleton objects which do not involve zero knowledge proofs.

  During presentation (or the presentation phase of an issuance), this first layer receives as input a presentation token description and choice of credentials; it outputs a mechanism specification, and a list of objects (zkModules) that will be needed for the proof and which will be sent to the second layer. This layer is responsible for assembling all the parameters (keys, configurations, etc.) required by the various building blocks, and is responsible for choosing an implementation for a given cryptographic primitive among the available alternatives.

  This layer delegates the generation of keys or parameters to the appropriate building block.

- The second layer consists of the so-called proof engine. It receives as input a list of zkModules from the first layer and generates/verifies a Fiat-Shamir zero-knowledge proof (signature of knowledge).

In the second part, the abstract building blocks only expose implementation-agnostic interfaces. The implementations thereof know how to perform a specific cryptographic primitive, but do not necessarily know how to interact with other primitives. In general, building blocks consist of one or more key/parameter generators, and of one or more proof interfaces (zkModules factories). The zkModules either (1) encapsulate the implementation of a given cryptographic primitive and expose a crypto-primitive-agnostic interface to the proof engine; or (2) have a structural role in the proof generation by indicating properties of attributes, relationships among attributes, or adding messages to be signed.

4.1 Parameter Generation

Let us describe the setup of parameters such as the system parameters, verifier parameters, or a key pair. This is a two-step process consisting of (1) creating a configuration template that has to be completed and submitted to (2) the generation of the set of parameters. First, the user of the Idemix library starts by requesting a new (default) configuration template from the crypto engine. The request is forwarded to the key generation orchestration singleton object. The orchestration element gathers all relevant data from the key manager before it queries all relevant building blocks for the implementation specific configuration parameters that need to be added to the template. The default configuration template is returned to the caller.

While a template is configured with default values that serve as suggestion for a general-purpose use, the actual settings must be set manually and in accordance with the planned use. Note, only the general entries as well as the entries corresponding to the chosen implementation must be filled out in the template. The entries corresponding to non-chosen implementations will be ignored. We call the completed configuration template a configuration.

The second step starts with the user passing the configuration to the crypto engine, which forwards it to the orchestration object. The latter, as in the creation of the configuration template, assembles the required parameters and forwards the data to the relevant building block corresponding to the chosen implementation. Finally, the building block generates the requested parameters upon which they are returned to the requester.

4.2 Presentation

Let us now illustrate the generation of a presentation token. When a user wants to create a presentation token it needs to pass the presentation token description, a list of credential URIs, and a list of pseudonym URIs to the crypto engine. These elements get forwarded to the proof orchestration object. The latter first fetches the credentials and pseudonyms based on their URI from the credential manager. Thereafter, it loads the system parameters, issuer public keys, credential templates, inspector public keys, and revocation authority public keys from the key manager. Then, it queries the building block factory for the building blocks required for the presentation token at hand. For building blocks that have several implementations, the proof orchestration may choose a specific implementation, or it may ask the building block factory for an implementation that is supported by the verifier.

These building blocks are used to generate a list of zkModules that serve as input to the proof engine. The proof orchestration object will configure each zkModule with the appropriate parameters such as the keys, credentials, or pseudonyms. The latter builds a way of encapsulating the state needed to perform part of the overall zero-knowledge proof. Note that zkModules responsible for proving ownership of a credential or a pseudonym may delegate part of the proof process to the smartcard manager. The proof engine generates a zero-knowledge proof supporting the validity of the presentation token based on this list of zkModules. Using the proof, the orchestration object updates the presentation token description and then combines the former with the zero-knowledge proof to form the final presentation token.

4.3 Verification

Matching the presentation token to a presentation policy needs to be performed before the former can be sent to the crypto engine for cryptographic verification. The crypto engine forwards the presentation token to the proof verification orchestration object, which fetches the relevant system parameters, issuer public keys, credential templates, inspector public keys, and revocation authority public keys from the key manager.

The verification orchestration needs to fetch the same set of building blocks from the building block factory as the prover did. To that end, the orchestration object uses the mechanism specification that describes the concrete implementations that have been chosen for each abstract building block. Thereafter, it can generate a list of zkModules using these building blocks, where the zkModules correspond to the list generated by the prover. This zkModules together with the zero-knowledge proof are sent to the proof engine for verification. The result of the verification is then forwarded to the calling entity.

4.4 Issuance

We describe the issuance process in the case where no jointly-random attributes are present and where the signing of the credential needs one round (as is the case for CL signatures). Note, the issuance protocol continues for as many rounds as the used signature building block specified and it reaches its end when the issuer specifies that it has sent the protocol ending message. In our description we follow the flow of the issuance messages, that is, beginning with the issuer we describe the process of the recipient creating an issuance token, the issuer verifying the latter and creating the signature and finally the recipient creating the credential.

4.4.1 Issuer: Create Issuance Policy

On the level of the Idemix library, the issuance process starts with the issuer invoking the crypto engine with an issuance policy and a list of issuer-set attributes, which are passed to the issuance orchestration object. The latter saves the issuance policy and list of attributes in the state storage, wraps the issuance policy in an issuance message, and returns that message to the upper layer.

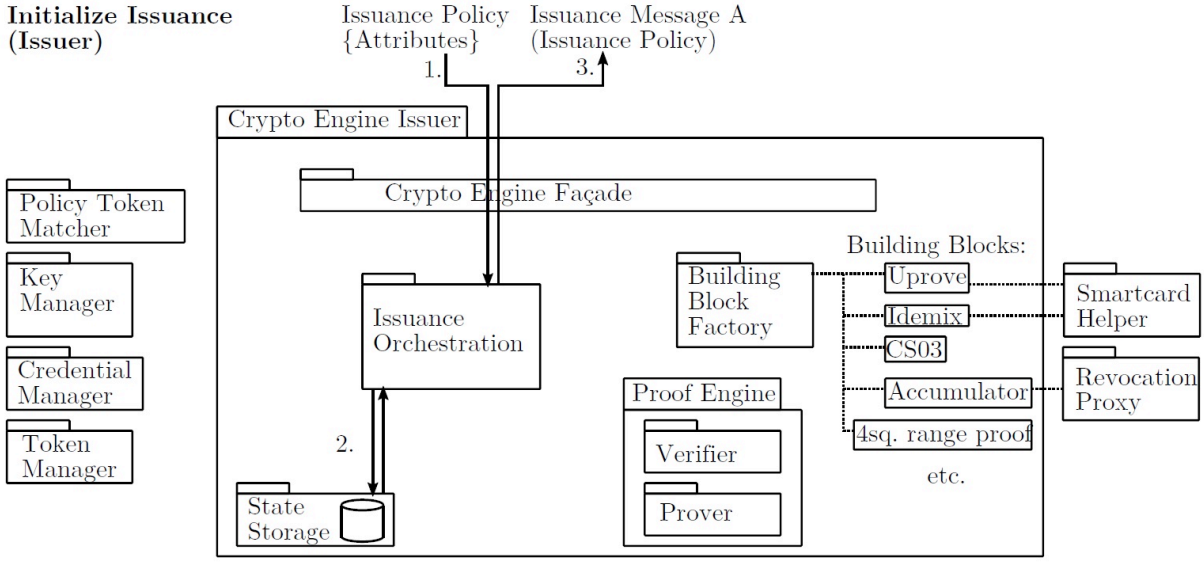

Figure 6. Initiation of issuance protocol on issuer's side.

The first step of the issuance protocol is shown in Figure 6. The issuer invokes the Crypto Engine with an Issuance Policy and a list of issuer-set attributes (1). The Crypto Engine forwards those to the Issuance Orchestration. The Issuance Orchestration saves the Issuance Policy and list of attributes in the State Storage (2), wraps the Issuance Policy in an Issuance Message, and returns that message to the issuer (3). The issuer should then transmit the message to the recipient.

4.4.2 Recipient: Generate Issuance Token

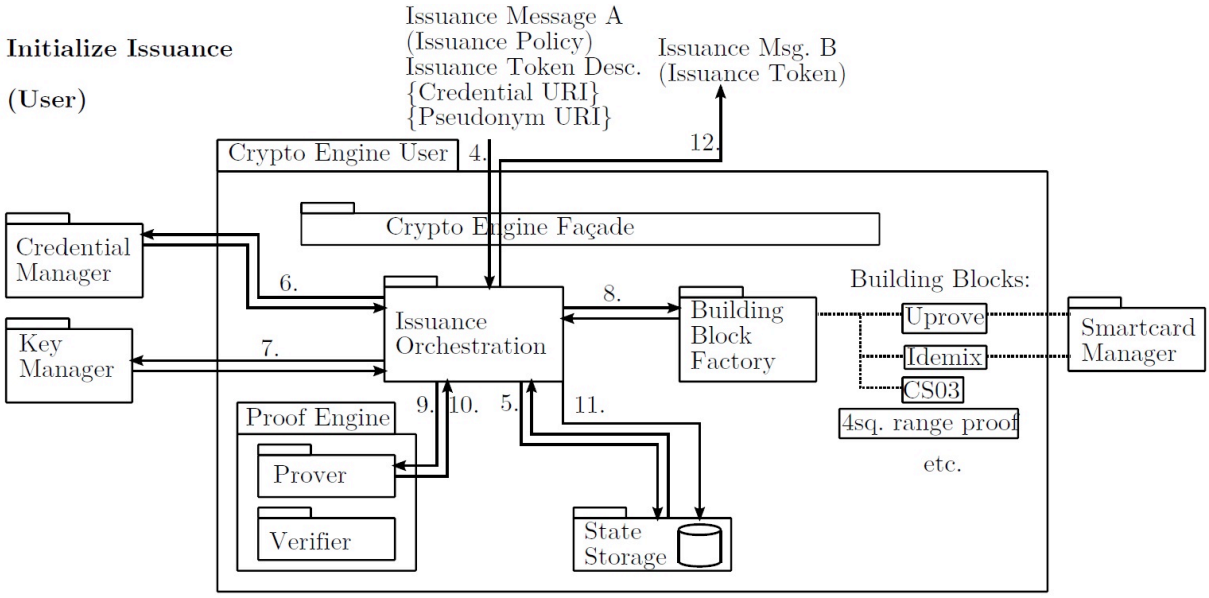

Figure 7. Recipient computes an Issuance Token with carry-over.

Figure 7 shows the recipient's side where the recipient calls the Crypto Engine with the Issuance Token Description, a list of credential URIs, a list of Pseudonym URIs, and the original Issuance Message (4). These elements are forwarded to the Issuance Orchestration. The latter first checks with the State Storage that the Issuance Context (contained in the Issuance Message) has never been seen before (5). Steps (6) to (10) are similar to a presentation proof, with the exception that the Issuance Orchestration additionally generates a zkModule from the signature building block that enables the carry-over of attributes. The Issuance Orchestration also retrieves state pertaining to the carry-over of attributes from the aforementioned zkModule after the Proof Engine finished the proof generation. It then completes the Issuance Token Description with data generated during the proof, generates an Issuance Token from the proof and the Issuance Token Description, wraps the Issuance Token in an Issuance Message, saves its current state in the State Storage (11), and returns the Issuance Message to the recipient (12).

4.4.3 Issuer: Create Signature

The third step of the issuance protocol is shown in Figure 8. The issuer should forward the recipient's Issuance Message (containing the Issuance Token) to his Crypto Engine directly. The latter forwards it to the Issuance Orchestration (13). The Issuance Orchestration first recovers the state associated with the Issuance Context from the State Storage (14). Then, it checks the proof contained in the recipient's Issuance Token similar to the presentation/verification-Orchestration (15)–(18). If the verification succeeded, the Issuance Orchestration then recovers the Revocation Authority's Public Key from the Key Manager (19), the issuer's Secret Key from the Credential Manager (20), and a Building Block for revocation of the correct implementation from the Building Block Factory (21). It then recovers state from the zkModule for carry-over and uses that state to initialize a zkModule for issuance from the signature Building Block; that zkModule is also initialized with the issuer-set attributes, and the Issuer Secret Key.

It also generates a zkModule for issuance from the Building Block for revocation. During creation time, the zkModule contracts the Revocation Authority (through the Revocation Proxy) to retrieve a new Revocation Handle and the associated Non-revocation Evidence. The Issuance Orchestration then

passes these two zkModules to the Proof Engine (22). During the proof generation, the zkModule for signature issuance will blindly sign the new credential. The Proof Engine returns the created zero-knowledge proof to the Issuance Orchestration (23). This zero-knowledge proof also contains the issuer's blind signature on the credential, the issuer-set attributes, the value of the Revocation Handle, and the Non-revocation Evidence. The Issuance Orchestration then queries the zkModule for signature issuance for the list of attribute values it knows about (including the revocation handle), and generates an issuance log entry containing that list. Finally, it wraps the zero-knowledge proof into an Issuance Message, and returns it to the issuer (24).

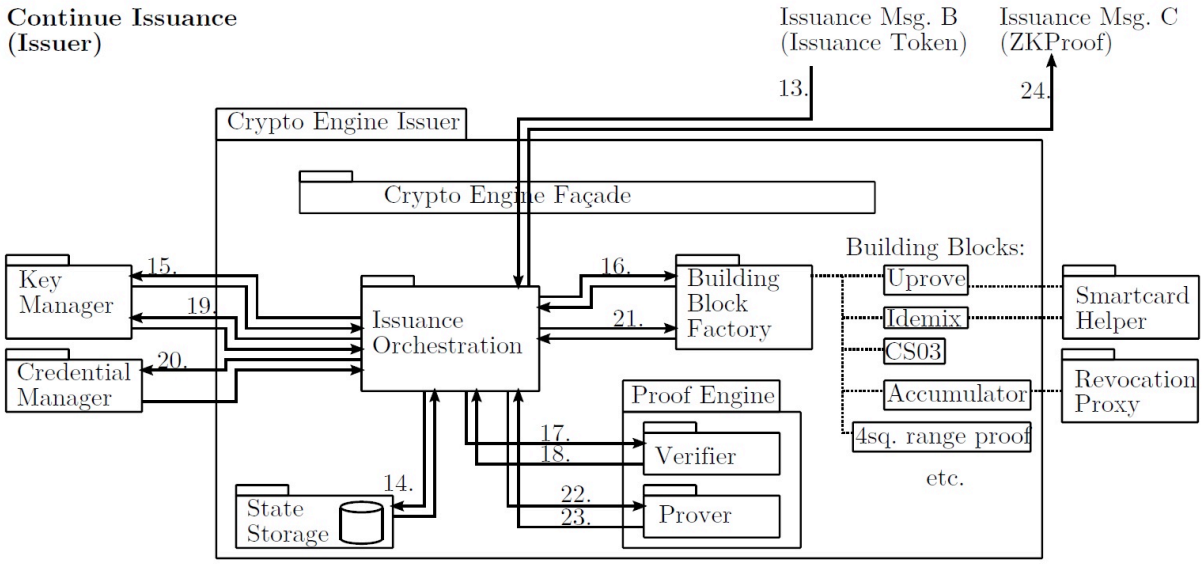

Figure 8. Issuer creates signature.

4.4.4 Recipient: Complete Signature

The issuer's issuance message (containing the zero-knowledge proof) is processed directly by the user's crypto engine by passing it to the issuance orchestration object. The latter recovers the state associated with the issuance context from the state storage. It retrieves the necessary parameters, specifications and keys from the key manager. Further, the orchestration loads a building block for signatures and a building block for revocation of the appropriate implementation from the building block factory.

The issuance orchestration gets a zkModule for signature issuance from the first building block, initializing it with its carry-over state and the issuer's public key. It also gets a zkModule for revocation issuance from the second building block. It then sends these two zkModules and the zero-knowledge proof to the proof engine for verification. After the issuance orchestration gets back the results of the proof verification, it extracts the issuance state from the zkModule for signature issuance. The issuance state can be combined with the carry-over state to recover the signed credential and non-revocation evidence. The signed credential is saved in the credential manager. The issuance orchestration returns the credential description to the upper layer.

5 Smartcard Implementation

ABC4Trust has defined a smartcard reference implementation referred to as *ABC4Trust Lite* to support the device-binding feature of Privacy-ABC systems. Throughout the project, the ABC4Trust Lite application has evolved in a number of ways; the first Patras pilot relied on version v1.0 based on a ZC7.5 BasicCard, whereas the Söderhamn pilot and the second Patras pilot have respectively made use of versions v1.1 and v1.2 based on a MultOS card with a larger non-volatile memory. The latest version (v1.2) of ABC4Trust Lite is the one put forward as part of the reference implementation. Its sources are publicly available on GitHub as well as its user manual.

5.1 Overview of ABC4Trust Lite

ABC4Trust Lite v1.2 is a dual-interface smartcard application that implements the device-binding versions of both U-Prove and Identity Mixer in a federated way, thereby supporting also other discrete-log-based, device-bound Privacy-ABC systems. The card also features a number of customized functionalities that were required by the pilots, such as counters and encrypted backups. These features can be easily removed or modified to fit other specific needs.

The rest of this section is dedicated to providing a technical insight on how ABC4Trust Lite v1.2 works and how it extends the ABCE by integrating it with a secure hardware device.

5.1.1 The smartcard platform

The target platform selected to implement ABC4Trust Lite v1.2 is a MultOS ML3 card with the following characteristics:

- The MultOS exact reference is ML3-36K-R1. The chip belongs to the Infineon SLE78 family and is equipped with a cryptoprocessor supporting modular and non-modular arithmetics.
- Available non-volatile memory (EEPROM): 64 KB,
- Available RAM: 1088 bytes (dynamic RAM) + 960 bytes (public RAM),
- Dual interface communications (contact T=0, T=1 and contactless T=CL),
- MultOS 4.3.1 Operating System running a MEL virtual machine and providing a number of native cryptographic APIs.

The sources of the application are written in ANSI C and compiled using the MultOS development tool chain.

5.1.2 The life cycle of a card

The card's life cycle is as follows:

Virgin mode. At delivery time, the card is in `virgin` mode. Its data memory is empty and the card is ready for personalization. Upon presentation of a 64-bit password via the dedicated application protocol data unit (APDU) command `SET ROOT MODE`, the card switches to `root` mode.

Root mode. Only in this mode can personalization be performed by initializing the various objects the card requires to run properly (4-byte global PIN, 8-byte PUK code, device private key, algebraic contexts, on-card issuers, settings of the on-card prover, possibly on-card credentials, etc). After personalization, sending a `SET WORKING MODE` command irreversibly switches the card to `working` mode.

Working mode. In `working` mode, the card interacts with issuers, verifiers and the card holder through the ABCE. The card holder has access to a number of PIN-protected commands that create, operate and remove on-card credentials.

Locked mode. The card will switch to `locked` mode if an incorrect PIN is presented 3 times in a row. It can be unlocked by presenting the personalized 8-byte PUK code.

Dead mode. The card falls into `dead` mode if an incorrect PUK code is presented 3 times in a row; a dead card is unusable.

5.1.3 High-level view of handled objects

The volatile memory of the smartcard is divided into two regions: the public memory and the dynamic memory. The public memory is used as in input/output buffer between the smartcard and the outside world. If more than one application resides on the smartcard, the public memory is also used to pass information from one application to another. In the simple case where there is only one application on the card, this zone is only used to pass APDUs between the application and the outside world. The dynamic memory is a private memory zone that cannot be externally read or written. If there is only one application on the card, then that application can read and write anywhere in the dynamic memory.

As depicted on Figure 9, the ABC4Trust Lite application manages a number of device-specific variables, several types of Privacy-ABC related objects and a free storage area called the BlobStore. Credentials point to their issuer; an issuer points to a group. The on-card prover is a stand-alone container that gathers the various data elements pertaining to proof sessions. All these objects can be created during personalization, except for the prover which pre-exits in memory and has to be configured so that proof sessions are managed appropriately. Credentials can be removed later by the card holder using the PIN, but the issuers and groups create during personalization will remain.

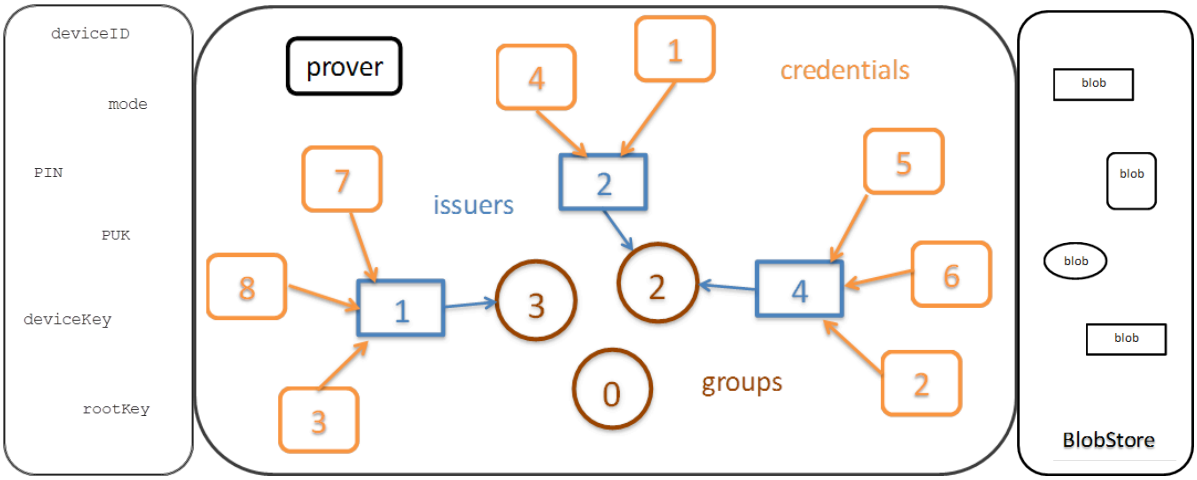

Figure 9. Persistent objects residing in the ABC4Trust Lite v1.2 non-volatile memory, including global variables (left), user-programmable objects (center), and the BlobStore (right).

These objects have the following purpose and meaning.

- **Algebraic contexts or groups:** a group is a collection of arithmetic parameters required to carry out algebraic computations. In addition to a modulus and optionally a group order and a cofactor, each group can also contain a number of group generators or bases. Privacy-ABC

systems defined over the same group but making use of different generators may therefore share the same group structure and just register their own generators to that group.

- **On-card issuers:** an on-card issuer is meant to reflect the existence of an external issuer. It is a container (i.e., a structure) that merely defines a working context for all credentials attached to it, namely: a group for conducting issuance and presentation of credentials, whether credentials should use two generators (as in Idemix) or just one generator (as in U-Prove), and how many times credentials can be presented before they expire. Several issuers may relate to the same group.
- **On-card credentials:** an on-card credential contributes to the issuance and presentation of a full-fledged credential handled by the ABCE. It can be invoked at issuance or presentation time to provide parts of zero-knowledge proofs (commitment and response) that serve as inputs to the full-fledged protocols operated by the ABCE. An on-card credential is attached to a unique on-card issuer.
- **On-card prover:** the on-card prover orchestrates proof sessions wherein one or several credentials are solicited to provide their commitment and response. The prover ensures that, once a proof session is open, the commitment of all involved credentials will share a common randomness. This allows the card to support proofs across multiple credentials. Pseudonyms and scope-exclusive pseudonyms can also be involved in proof sessions.

Credentials can be managed directly by the card holder as they are just PIN-protected. All the other Privacy-ABC related objects can only be created and stored when the card is in `root` mode.

- **Blobstore:** the Blobstore is a PIN-protected free storage space allocated by the card. The Blobstore manages data under the form of key-value associations -- very much like in associative arrays -- that are referred to as blobs. The application provides commands to store, update, read and remove blobs from memory. The Blobstore is used by the ABCE to store full-fledge credentials and other user-related parameters directly on the card. The card does not attempt to interpret the contents of the Blobstore.

5.1.4 Card personalization

Incoming and outgoing data (in the usual card-centric terminology) are transmitted back and forth between the card and the terminal application. The card allows write access to an internal input buffer. The card supports extended APDUs, so that arbitrary byte streams of up to 512 bytes can be written into the buffer to provide data material to the card. This is done using the APDU command `PUT DATA`.

Personalization is performed in three stages:

Stage 1 (root authority): the card is switched to `working` mode by sending a `SET ROOT MODE` command containing a 64-bit password. An RSA public key, referred to as the root key, must then be provided to the card using the `SET ROOT KEY` command. The root key serves as a means for the card to public-key encrypt data to the root authority in the case where the rest of the personalization is performed by a third party. The root authority makes use of the private part of the root key as a decryption key. The encryption scheme is detailed in the documentation on GitHub.

Stage 2 (third party): the command `INITIALIZE DEVICE` initializes the PIN and PUK codes to random values and returns them encrypted under the root key. This allows the delegation of personalization by the root authority to some untrusted third party without endangering the confidentiality of the card holder's PIN and PUK codes. `INITIALIZE DEVICE` also initializes the device private key (that never leaves the card), various size parameters and a device identifier the access to which will be PIN-protected in `working` mode.

Stage 3 (third party): the personalizer plays a series of APDU commands that create groups and on-card issuers, configure the on-card prover and possibly store blobs and/or create on-card credentials in the card's non-volatile memory. When the personalizer is done programming all Privacy-ABC related objects, the command `SET WORKING MODE` is played to put the card in `working` mode.

5.2 Instantiating U-Prove and Identity Mixer

Instantiating U-Prove. U-Prove makes use of groups of known order and requires only one generator, which amounts to defining a prime modulus m = p, a group order q, a cofactor f and a single generator g. U-Prove issuers are therefore created in a single-base setting at personalization time.

Instantiating Idemix. Comparatively, Idemix makes use of groups of unknown order and requires two generators g_1, g_2. Idemix issuers are personalized accordingly, i.e., in a double-base setting.

Instantiating other Privacy-ABC systems. Our applicative architecture allows supporting other device-bound Privacy-ABC systems. Those can be categorized according to their algebraic context (known or unknown order) and whether issuers are in the single-base or double-base setting. ABC4Trust Lite supports the 4 categories of Privacy-ABCs shown on Table 8.

Table 8. Device-bound Privacy-ABCs supported by ABC4Trust Lite.

Group of	**known order**	**unkown order**
Single-base issuers	U-Prove	also supported
Double-base issuers	also supported	Identity Mixer

5.3 Potential Extensions

A possible extension consists in supporting elliptic curve operations in addition to multiplicative integer groups. To this end, one may enrich the format of group containers to include components for the curve parameters a and b (classical groups would leave these two components empty), and use the modulus to store the field characteristic. The components for the group order q and co-factor f would be unchanged. Group generators would now store either integers or points on the curve, using some format for parsing the point coordinates (concatenation would do). The low-level arithmetic API would then automatically select the appropriate operations depending on the nature of the algebraic settings.

Another approach is to embed full-fledged Privacy-ABC systems in the card. This would be a major change as the application would then manage attributes locally and more generally would support more or less the same services as the ABCE. This means that the card must be able to parse XML security policies (maybe a more card-friendly format can be defined to express policies in that context), perform credential matching on its own and handle the issuance and presentation of full-fledged credentials. Such an application (say, ABC4Trust Pro) would be far more intricate to design and implement efficiently; it would however also be much more powerful.

6 Demonstration of the Reference Implementation

In order to demonstrate the features of the reference implementation, a small demonstration scenario has been implemented and deployed in a virtual machine for easy distribution. The virtual machine image, as well as a zip archive containing the various services, can be obtained from the ABC4Trust website (https://abc4trust.eu/demo/hotelbooking). We encourage the reader to open and experiment with the software installed in the virtual machine. Further instructions are below.

The scenario showcases most of the features provided by reference implementation, namely issuance with key carry-over, verification and revocation of credentials. Inspection and issuance with carry-over attributes are not included in the demonstration, but could be included in a future version.

The scenario is based around the case of booking a hotel room. In order to book a room, a user must possess valid passport and credit card credentials; however the user will be forced to reveal neither her identity nor her credit card number.

A potential feature to add in a future version of the demonstration would be to make the credit card number in the presentation token inspectable in case of a (late) cancellation. This would allow the hotel to withdraw a fee from the customer if certain criteria are met. Similarly, an additional issuer of student card credentials could be implemented. This could be used to showcase advanced issuance (where the student's attributes are carried over from his student card to the credit card credential) as well as the verifier having different presentation policies, offering students a discount.

6.1 Content of the demonstration

The demonstration consists of the following services:

- A Grails application running the webpage for a hotel, acting as the verifier.
- A Grails application running the webpage for a bank, acting as the issuer of credit card credentials.
- A Grails application running the webpage of governmental agency, acting as the issuer of passport credentials.
- A revocation service, able to revoke credit card credentials.
- A user service, able to manage credentials, communication with smartcards and perform presentations of credentials.
- A user UI service, providing a GUI for the user service.

6.2 Deploying the demonstration

The demonstration is deployed as a virtual machine. We have verified it to work with VMware Player 5, or VirtualBox version 4.1.8. It should work equally well with VMware Fusion 5 and 6 or comparable versions of VMware Workstation. To use the virtual machine one should open the image and boot the virtual machine.

6.2.1 Deploying the virtual machine

After booting the virtual machine, simply log on with the password: "**abc4trust**".

The virtual machine already has all the required dependencies installed and is running the various services. If something should go wrong, each service can be started by running the appropriate shell script found in *~/ABC4Trust_demo*. E.g. In order to start the credit card issuer, go to *ABC4Trust_demo* and run the command *./start-creditcard-issuer.sh*. The user service is using a pre-generated software smartcard with pin 1234. If you wish to use a hardware smartcard instead, refer to 6.4 on how to initialize and use the hardware smartcards in connection with the user service.

6.2.2 Deploying the zip-archive

The zip archive available on the website (https://abc4trust.eu/demo/hotelbooking) along with the virtual machine image contains a prebuilt Firefox plugin, a copy of the resources used in the demo, a hardware smartcard initialization application and the 6 previously mentioned services as well as a copy of certain maven repository files. Maven is required to run the various services; it can be obtained from *http://maven.apache.org/*

- Start by extracting all the files to some folder.
- Then copy the contents of the *repository* folder to your Maven repository folder (typically in *~/.m2/repository* in Linux)
- Next install the browser plugin, by browsing to the *abc4trust.xpi* file using Firefox and following the instructions.
- You should now be ready to run the services. Each service is started by opening a command prompt, browsing to the appropriate folder and running one of the following commands:

Service:	Folder:	Command:
Revocation authority	revocation-authority/	java -jar selfcontained-revocation-service.war
Hotel verifier	demo-verifier-hotel/	mvn grails:run-app -Dserver.port=8080
Passport issuer	demo-issuer-cpr/	mvn grails:run-app -Dserver.port=8181
Creditcard issuer	demo-issuer-creditcard/	mvn grails:run-app -Dserver.port=8282
User service (with software smartcard)	user-client/idselect-user-service/	mvn -Dtestcase=hotelbooking -DUserSoftwareSmartcard=HotelBookingSC jetty:run
UI service	user-client/	mvn -pl eu.abc4trust.ri.ui.user.product jetty:deploy-war

6.3 Using the demonstration services

The objective of the demonstration is to book a hotel room, however in order to do so, you must possess valid credit card and passport credentials. In the following we assume all the services are running, which can be achieved by following the instructions in Section 6.2

6.3.1 Obtaining a passport credential

We start by issuing a passport credential by visiting the webpage for issuer. The webpage is running on *http://localhost:8181/demo-issuer-cpr/* , if you are using the virtual machine image, a browser tab with the webpage will already be open.

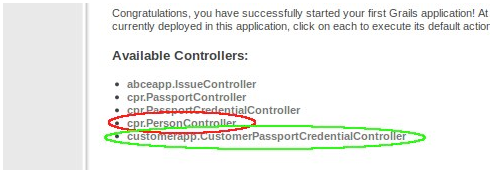

Figure 10. Start screen of the demo.

The issuer maintains a database of people eligible for a passport credential. In order to be able to get a credential issued, you will have two options. Either you can use a preexisting credential, or you must create a new person, attach a passport to this person and finally attach a credential to the passport.

To create a new person and attach a credential, start by following the link *"Cpr.personController"* (the green circle in Figure 10. Start screen of the demo.) and choosing *"Create new person"* on the new webpage.

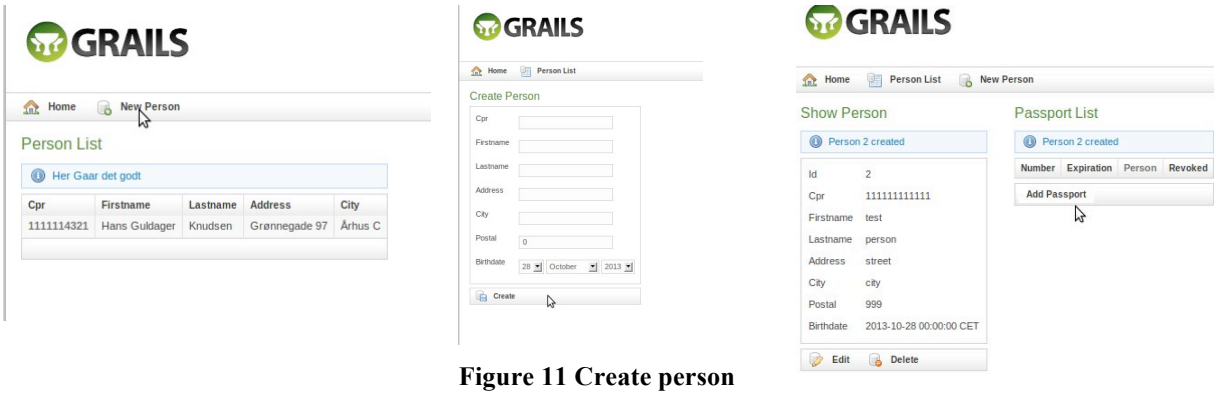

Figure 11 Create person

After filling in the details and clicking *"Create"* (as seen in Figure 11 Create person), you will be redirected to a webpage showing the newly created person. On this webpage, you should have the option to click on *"Add Passport",* which will create a new passport in the *Passport List.* If you click on the newly created passport, you will be redirected to a similar page, where you have the option of creating a credential for the passport, as can be seen in Figure 12 Add passport credential. To do so, simply click *"Add PassportCredential"*, which will enable issuance of credentials for the newly created passport. At this point, you should write down the values for **cpr, (passport) number and issuance key**, as they are the required values for issuance.

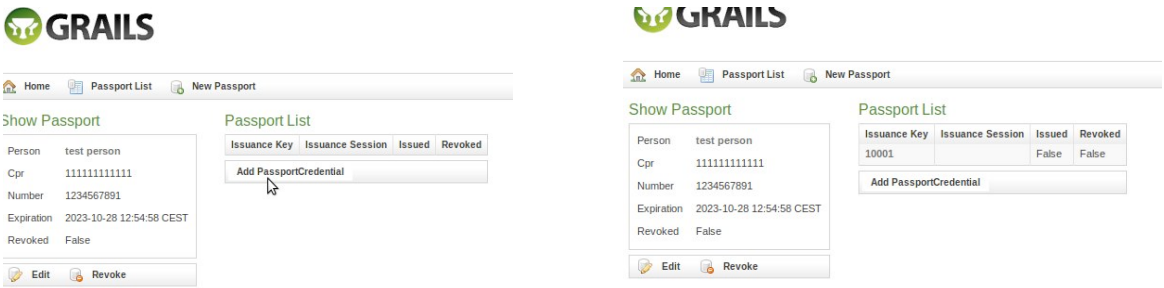

Figure 12 Add passport credential

As mentioned earlier, it is also possible to use a preexisting credential. In this case, use the following values:

Cpr.:	1111114321
Passport number:	1234567890
Issuancekey:	10000

Now you are ready for the actual issuance. On the front page, click on the link "*customerapp.CustomerPassportCredentialController" (the red circle in* Figure 10. Start screen of the demo*).* This will take you to an issuance page where you start the issuance by filling in the fields and clicking *"Issue"* as seen in Figure 13 Start issuance.

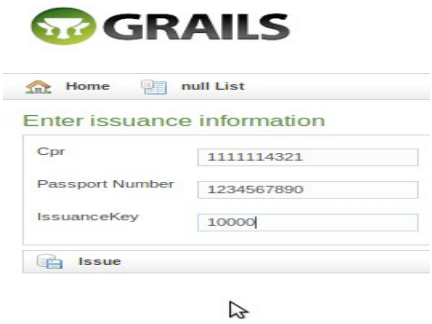

Figure 13 Start issuance

The ABC4Trust Firefox plugin will now open a window, asking for a PIN for the smartcard. If you are using the pre-generated smartcard, the code is "1234".

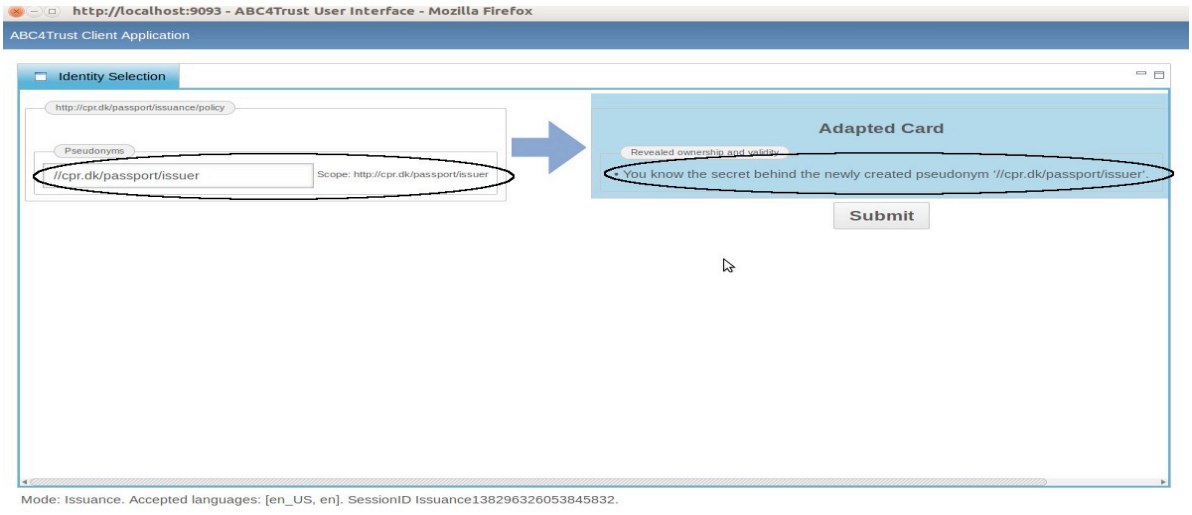

Figure 14 Client application

Another window will now open. This is the ABC4Trust Client Application (Figure 14 Client application), which takes additional user inputs and shows what information will be revealed. In this case we will create a new pseudonym, which have the option to give a special name or simply use the default value *"//cpr.dk/passport/issuer"*. Once ready, you should click *"Submit"*, which will close the window and start the issuance protocol. The user ABCE will now communicate a bit with the issuer.

Once they are done with the issuance protocol, the webpage should say *"Credential obtained... SUCCESS"* and you will have obtained a passport credential.

6.3.2 Obtaining a credit card credential

The process of issuing a credit card credential is similar to issuing a passport credential:

You start by visiting the bank's webpage running at http://localhost:8282/demo-issuer-creditcard/. As with the passport issuance, a browser tab should already be open if you are using the virtual machine.

Again it is possible to either create your own user or use a pregenerated person. The process of creating a custom person and credit card is identical as the one described in Section 6.3.1, only substituting passport with credit card.

The pre-generated person has the following values:

Cpr.:	1111114321
Creditcard number:	1111222233330000
Issuancekey:	20000

Next step is clicking on the link *"customerapp.CustomerCreditcardCredentialController*" and fill in the chosen values. Again the ABC4Trust Client Application should open and issuance commence.

Once you see the *"Credential obtained... SUCCESS"* message, issuance is complete.

6.3.3 Booking a hotel room

Having obtained both credentials, you are now able to book a room by browsing to the hotel booking page, located at http://localhost:8080/demo-verifier-hotel/booking/index. As in the previous sections, if you are using the virtual machine image a browser tab should already be open. On this page, you are able to fill out the criteria for your room and search for available rooms. Once you have found your desired room, click *"Book this room"*, which will open the ABC4Trust Client Application window again.

As in the previous sections, here you can see what is about to be revealed to the hotel. In this case, you are only revealing that you own a passport and credit card credential from certain issuers.

If you had issued multiple credentials of the same type, the menu to the left would have allowed you to choose between the combination(s) of credentials satisfying the hotel's presentation policy.

Once you are happy with your choice, click *"Submit"* and the user ABCE will create a presentation token and send it to the hotel. Once the hotel verifies the token, the webpage should say *"Verification was successful"* and you booking has been accepted.

6.3.4 Revoking a credential

The passport credential is revocable. In order to revoke the credential, the credential's revocation handle (as XML) must be posted (via a HTTP POST request) to the revocation authority webservice. The sample file RevocationHandle.xml must be edited to contain the revocation handle (the handle itself can be found in the manage credentials menu in Firefox) desired for revocation.

Once the file has been saved, it can be posted. In Linux, this is done using the following curl command:

```
 curl -X POST --header 'Content-Type: text/xml' -d @RevocationHandle.xml
http://localhost:9500/revocation/revoke/http%3A%2F%2Fcpr.dk%2Fpassport%2Fspecificat
ion%2FrevocationUID'
```

When using the virtual machine image, the command is conveniently stored as a shell script which can be executed running the command ./revoke.sh from the ABC4Trust_demo folder.

6.4 Hardware smartcards

The demonstration also supports the use of hardware smartcards. It is assumed that the smartcard is new, in the sense that it has just been loaded with a new ALU, namely abcmultos_v18_soder.alu.

First make sure the smartcard is in the reader and any necessary drivers in installed.

Start by extracting the *HotelBookingResources* folder from the zip archive. Next open a command prompt and browse to *HotelBookingResources* folder. You initialize the smartcard by running the command "*java -jar initializeHardwareSmartcard.jar . 42"*. This will install the necessary parameters to the smartcard and protect it with a PIN-code. The PIN can be found in the newly created folder "*sc_output"*.

In order to run the user service using the hardware smartcard, follow the instructions from Section 6.2, only using the command "*mvn –Dtestcase=hotelbooking jetty:run"* when starting the user service.

7 Performance and Timings

The performance of the reference implementation has received substantial attention because the pilot in Söderhamn in particular was showing slow performance on older hardware. Performance issues were reported from the pilot running on the school in Söderhamn, Sweden. Running the user-client on a computer comparable to the old computers (Single core CPU, 1 GB RAM) used at the school showed that a simple presentation could take up to 30-40 seconds. Benchmarks of the user client showed a couple of bottlenecks:

1. At each presentation, a credential's revocation information is updated, causing the credential to be deleted from the smartcard and then stored again. This took about 3.5 seconds per credential.
2. Calculations in the crypto engine were slow on the school computers.

Some improvements have been suggested:

1. Reduce the key size from 2048 bits to 1024 bits.
2. Make all credentials except the main credential (which is used in all presentations) non-revocable.
3. Only update a credential on the smartcard when it has been revoked.

Reducing the size of the keys would have improved the performance drastically - a benchmark showed a 43% speed improvement on a presentation, but it has not been possible to implement this in the Söderhamn pilot due to a limited timeframe. The two other suggestions were implemented in the Söderhamn pilot, all three were implemented for the Patras pilot.

7.1 Experimental Results

We performed our efficiency analysis using an Intel Duo Core 2.2GHz machine with 2GB of RAM running Windows 7 (32 bit). This slightly aged configuration was chosen on purpose to be comparable to the typical infrastructure at the Söderhamn school. The hardware smartcard was a contact/contactless dual-interface MultOS ML3 running the MultOS 4.3.1 operating system. The card is equipped with an Infineon SLE78 chip with cryptocoprocessor, 64 KB of non-volatile memory (EEPROM), 1088 bytes of dynamic RAM, and 960 bytes of public RAM. The tests were performed with JProfiler 8.0.1 as a unit test run through Eclipse 4.2.2, with all entities (user and verifier) running on the local machine.

Table 9 and Table 10 gives an overview of the time spent for different parts of a presentation protocol for modulus sizes of 1024 and 2048 bits, respectively. We display the information for a "fresh" presentation, meaning, including the initialization of the engine and loading information such as group parameters and credentials from the smartcard. Subsequent presentations in the same session would be considerably faster since this information is cached on the PC.

The tables display number for different sorts of presentations, depending on the number and type of credentials involved. The first column involves no credential at all but merely proves possession of a pseudonym. The second shows numbers for a single U-Prove or Identity Mixer credential, the third and fourth for two U-Prove and two Identity Mixer credentials, and the fifth for a mixed presentation token that is derived from two credentials, one of which is a U-Prove credential and the other of which is an Identity Mixer credential. Note that the presentations involving credentials do not present any pseudonyms. Figure 15 and Figure 16 display the same information in graphical form for better visualisation.

Table 9. Timing results for presentation with 1024-bit moduli (in ms).

	Pseudonym only	1 U-Prove	1 Idemix	2 U-Prove	2 Idemix	1 U-Prove, 1 Idemix
Smartcard operations	4224	3562	5524	6172	10117	8297
Getting data from smartcard	6256	6266	6348	7051	6559	7573
Getting group parameters from smartcard	434	1059	1249	1660	1882	1784
ABCE	815	1442	1193	1525	1389	1354
XML parsing	8	93	107	104	111	119
Other	200	931	708	878	887	783
Total	**11937**	**13353**	**15129**	**17390**	**20945**	**19910**

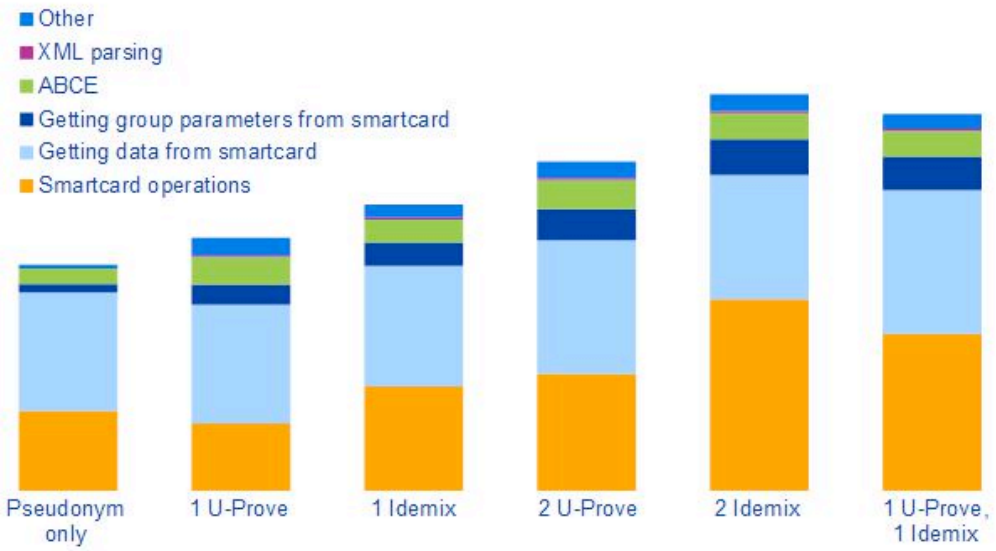

Figure 15. Timing results for presentation with 1024-bit moduli.

Table 10. Timing results for presentation with 2048-bit moduli (in ms).

	Pseudonym only	1 U-Prove	1 Idemix	2 U-Prove	2 Idemix	1 U-Prove, 1 Idemix
Smartcard operations	7874	4564	6222	8129	11310	9546
Getting data from smartcard	7399	7393	7169	7774	7446	8415
Getting group parameters from smartcard	758	1640	2343	2562	2853	2454
ABCE	1806	2323	1906	3358	2499	3085
XML parsing	15	104	85	148	116	104
Other	500	998	694	876	780	772
Total	**18352**	**17022**	**18419**	**22847**	**25004**	**24376**

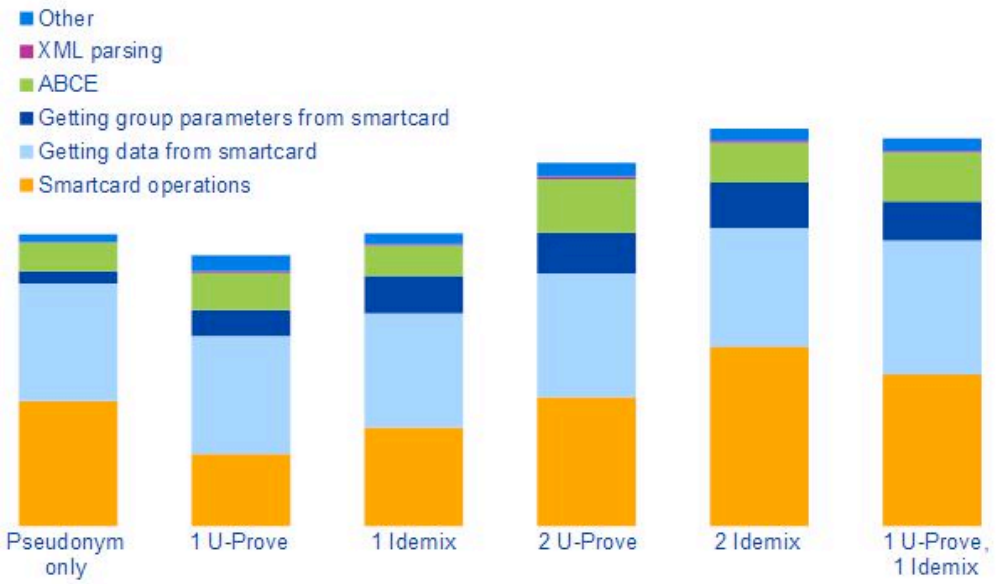

Figure 16. Timing results for presentation with 2048-bit keys.

7.2 Discussion

From the experimental results, one can tell that the smartcard is quite a performance bottleneck. About half of the time spent in a presentation is spent on transferring the group parameters and other data from the smartcard. Once fetched from the card, however, this data is cached on the PC, so that this delay would not appear anymore in subsequent presentations using the same smartcard. Performance could also profit from more compact credential representations, which are currently not optimized for size.

Cryptographic computations on the hardware smartcard (called "smartcard operations" in Figure 16) also take a considerable fraction of the time, between a quarter and half of the time, depending on the setting. Overall, between 80% and 90% of the time spent is related to the smartcard, which seems to indicate that there is room for improvement there. On the good side, the delays of these operations are

nearly reduced to zero when the hardware smartcard is replaced with emulation in software, which is what one would do in a software-only setting.

Apart from smartcard operations, in an actual deployment the network delays when communicating with the verifier, issuer, or the revocation authority can cause additional delays. Also, the identity selection GUI fetches and caches a graphical representation for each credential, which may incur some extra overhead the first time a credential is used.

The operations on the PC take a relatively small amount of time, as do the software cryptographic routines and the handling by the ABCE layer.

8 Conclusion

This deliverable provides a status update on the reference implementation. Solid progress has been achieved by making the source code of the new cryptographic layer as well as the smartcard code available.

The rewritten cryptographic architecture has only slightly impacted the API of the ABCE layer, showing the versatility of the current ABCE architecture and implementation. Other parts of the ABCE have seen improvements as well improving the stability, performance, and usability of the components.

In terms of efficiency, presentation of a 2048-bit credential takes roughly between 17 and 25 seconds. Most of this time, however, is spent on the smartcard, either for transferring data or for performing cryptographic operations. The good news is that these times vanish almost completely when using the ABC4Trust stack in a software-only setting, so that one can expect presentation times around 3-5 seconds.

9 Bibliography

[BCE13] Patrik Bichsel, Jan Camenisch, Robert R. Enderlein, H2.3 Specification of the Identity Mixer Cryptographic Library, ABC4Trust Heartbeat, 2013.

[Bra99] Stefan Brands. Rethinking Public Key Infrastructure and Digital Certificates— Building in Privacy. PhD thesis, Eindhoven Institute of Technology, Eindhoven, The Netherlands, 1999.

[CL01] Jan Camenisch and Anna Lysyanskaya. Efficient non-transferable anonymous multi-show credential system with optional anonymity revocation. In Birgit Pfitzmann, editor, Advances in Cryptology — EUROCRYPT 2001, volume 2045 of Lecture Notes in Computer Science, pages 93–118. Springer Verlag, 2001.

[BCD+14] Patrik Bichsel, Jan Camenisch, Maria Dubovitskaya, Robert R. Enderlein, Stephan Krenn, Ioannis Krontiris, Anja Lehmann, Gregory Neven, Janus Dam Nielsen, Christian Paquin, Franz-Stefan Preiss, Kai Rannenberg, Ahmad Sabouri, Michael Stausholm, “D2.2 Architecture for Attribute-based Credential Technologies - Final Version”, 2014.

[GN12] Hans Guldage, Janus Dam Nielsen. “Initial Reference Implementation”, ABC4Trust Deliverable D4.1.