Integration of hardware tokens in the Idemix library[⋆]

Antonio de la Piedra

Radboud University Nijmegen, ICIS DS, Nijmegen, The Netherlands
`a.delapiedra@cs.ru.nl`

Abstract. The Idemix library provides the implementation of the Camenisch-Lysyanskaya (CL) Attribute-based Credential System (ABC), its protocol extensions and the U-Prove ABC [3, 7]. In the case of the CL ABC, the library can delegate some cryptographic operations to a hardware token (e.g. a smart card). In the last few years several practitioners have proposed different implementations of ABCs in smart cards [1, 2, 10, 15, 17]. The IRMA card provides at the time of writing this manuscript, an optimal performance for practical applications. In this report, we address the case of integrating this implementation in the Idemix library. We opted for implementing the key binding use case together with the generation of exclusive scope pseudonyms and public key commitments on card. The integration requires two additional classes (one that parses system parameters, credential specifications and issuer public keys and other one that interfaces the card and its functionalities with the CL building block) together with one modification in the code if the signature randomization is delegated to the card (only required in one of the proposed alternatives). The integration of the key binding use case requires 540 bytes extra in the smart card. We can perform all the involved cryptographic operations in only 206.75 ms, including the computation of exclusive scope pseudonyms (55.19 ms).

[⋆] The work described in this paper has been supported under the ICT theme of the Cooperation Programme of the 7th Framework Programme of the European Commission, GA number 318424 (FutureID). This work was carried out while visiting the IBM Zurich Research Laboratory (July 2014).

Table of Contents

1 Introduction

This manuscript describes the integration of the IRMA card[1] in the Idemix 3 library[2]. This version integrates the U-Prove credential system [3] together with Idemix [7, 8] and its extensions e.g. for performing range proofs and verifiable encryption [4, 9].

The library implements the different functionalities that are present in any ABC system for provers, verifiers, revocation agents and issuers [6]. For instance, the CL issuing protocol, involving a prover and a certain issuer generates a CL signature over a set of attributes in a credential form. This can be used later for performing proofs of ownership between a prover and a certain verifier in the verification protocol.

The library assumes issuers with public keys generated according to a set of public system parameters and users equipped with credentials that can be used for proving properties of their attributes and generate exclusive scope pseudonyms [7].

Recently, Vullers adapted the Idemix 2.3.4 library to the IRMA card by patching different parts of the implementation. This modification serves for issuing credentials to the smartcard and verify proofs generated by IRMA[3]. That means, that the Idemix library is solely used for constructing terminal verifiers and issuers. However, the new version of Idemix, fundamentally based on design patterns, provides new opportunities for integrating hardware implementations

[1] `https://www.irmacard.org/` (accessed August 15, 2014)
[2] `https://prime.inf.tu-dresden.de/idemix/` (accessed August 15, 2014)
[3] `https://github.com/credentials/idemix_library`, (accessed September 3, 2014)

of Attribute Based Credentials (ABCs) in the library by delegating certain operations to the card whereas a certain deployment can be consistent with the ABC4Trust scheme for policy specification [5]. Moreover, by delegation only certain operations to the card, it is possible to perform complex proofs in the cardholder's workstation. Otherwise, that could take a considerable amount of time in performance and RAM-constrained embedded devices. All in all, the disadvantage of this approach is the non-standalone nature of the card, depending of an external workstation where the credentials, issuer public keys and system parameters are stored.

Currently, only the key binding use case is integrated. Nonetheless, the card can also randomize a triple (A, e, v) of a certain credential, compute domain pseudonyms and public key commitments orchestrated from the extended secret manager of the library. This integration comprises three parts:

1. A Java implementation that interfaces the IRMA card using the smartcard API of Oracle Java[4]. It also parses a certain description of the system parameters, the specification of a credential and an issuer public key and uploads those parameters to the smart card.
2. A smart card implementation that relying on the arithmetic methods of the IRMA card supports the requirements of the new Idemix 3 architecture.
3. Only one modification in the current implementation of Idemix 3 related to the size of the v component of the CL signature is required if signature randomization is performed on the card (Section 3.4).

In the next section, we briefly explain the functionalities of the IRMA card and describe the required modifications in order to integrate it in Idemix. Then, we identify the structures that must be extended and the requirements for this hardware token. In Section 3, the architecture of Idemix 3 and their functionalities is described and we propose three different alternatives for integrating the card. In Section 4 we explain the changes performed for integrating the key binding use case. We describe the performance figures of the IRMA card within the key binding use case in Section 5 and end in Section 6 with some conclusions.

2 The IRMA card

The IRMA card implements a subset of Idemix [6], particularly the issuing protocol and the selective disclosure operation. In contrast to prior attempts in the literature to implement those primitives [2, 16] in a reasonable time, IRMA performs the selective disclosure over credentials of 5 attributes in 1–1.5 seconds according to the number of revealed attributes [17].

Each credential acts as an attribute container, protected by a CL signature generated by an issuer. This signature guarantees the integrity of the credentials. For instance, modification or deletion of new attributes can be easily detected

[4] `http://docs.oracle.com/javase/7/docs/jre/api/security/smartcardio/spec/javax/smartcardio/package-summary.html` (accessed August 15, 2014)

by a verifier. Moreover, each credential is connected to the cardholder by her master secret, securely stored on the card.

After the issuing process, the users owns a CL signature over one credential. This is represented by the triple (A, e, v) over $l = 5$ attributes in the case of IRMA: $(m_0, m_1, ..., m_5)$. We consider m_0 the cardholder's master secret. This information is stored in the card for each credential. The CL signature is created by an issuer according to its public key[5] $(S, Z, R_0, R_1, ..., R_5 \in QR_n, n)$ using its secret key (p, q). For instance, a CL signature over a set of attributes $(m_0, ..., m_5)$ is computed by selecting A, e and v s.t. $A^e = ZR_0^{-m_0} R_1^{-m_1} R_2^{-m_2} R_3^{-m_3} R_4^{-m_4} R_5^{-m_5} S^{-v}$ mod n. Then, a third party can check the validity of the signature by using the issuer's public key and the tuple (A, e, v) as $Z \equiv A^e R_0^{m_0} R_1^{m_1} R_2^{m_2} R_3^{m_3} R_4^{m_4} R_5^{m_5} S^v$ mod n. In IRMA, for performance reasons, the size of the modulus n is restricted to $l_n = 1,024$ bits whereas the attributes are represented as $l_m = 256$ bits. The rest of parameters are set as $l'_e = 120$, $l_\o = 80$, $l_H = 256$, $l_e = 504$, and $l_v = 1,604$ bits[6].

The crucial property of the CL signature in Idemix is to prove its possession without revealing additional information. Moreover, one can also perform the selective disclosure of the cardholder's attributes via discrete logarithm representation modulo a composite proofs of knowledge [12].

In IRMA, the prover part of Idemix is implemented in the card as a set of states (`PROVE_CREDENTIAL`, `PROVE_COMMITMENT`, `PROVE_SIGNATURE` and `PROVE_ATTRIBUTE`) that maps the prover and verifier interaction between a terminal and the smart card[7]. In each transaction, both entities exchange ISO 7816 APDUs that retrieve and write data in the smart card volatile (RAM) and non-volatile (EEPROM) memories [14]. When the card receives a verification request, it changes its initial state to `PROVE_CREDENTIAL`. Then, it acquires a presentation policy with the description of the attributes that must be revealed. Then, the card performs the randomization of the signature and the generation of the corresponding *t_* and *s_values* (`PROVE_COMMITMENT`). Afterwards, the card changes its working state to `PROVE_SIGNATURE`. In this state, the verifier can request the randomized tuple $(A', \hat{e}, \hat{v}')$. Finally, the card switches to `PROVE_ATTRIBUTE`, where the verifier is allowed to request the set of revealed and hidden attributes related to the proof.

As we note in the next section, this collection of states cannot be directly reused in the integration of the card in Idemix 3.

[5] The Idemix 3 library also considers the bases R_t, R_d and R_s, where the latter is equivalent to S.

[6] The term l'_e represents the size of the interval where the e values are selected, $l_\o$ is the security parameter of the statistical zero knowledge proof, and l_H is the domain of the hash function used in the Fiat-Shamir heuristic (we use SHA-256). Finally l_e and l_v are related to the size of e and v parameters of the CL signature.

[7] We refer the reader to [17] for a description about how a (A, e, v) triple is obtained by the card.

3 The Identity Mixer Cryptographic Library

In this section, we describe the architecture of the Idemix 3 library and introduce the main building blocks that can be extended for integrating a hardware implementation of Idemix, in this case, the IRMA card.

Since the integration of the card is based on the prover operations, we restrict ourselves to the description given to this entity in the Idemix specification.

In the proving protocol, a certain verifier selects which restrictions the prover must fulfill related to their attributes and its values (i.e. this is specified in the presentation policy, following the ABC4Trust terminology [5]). Then, the prover generates a cryptographic proof that validates the restrictions imposed by the verifier (i.e. the presentation token). This token is generated by the crypto engine of the library via the list of the credentials involved and the pseudonyms of the prover. In the next section, we describe the main structures that are responsible of generating the presentation token.

3.1 ZkModules, the proof assembler and the proof engine

The two main components of the Idemix library are the proof assembler and the proof engine. The proof assembler relies on the cryptographic building blocks of the library for generating a list of zero knowledge modules (ZkModules) that perform partial zero knowledge proofs according to a presentation description sent by a certain verifier to the prover. This component receives a presentation token description together with a list of credentials and pseudonyms involved in the proof.

The proof assembler fetches the required credentials and pseudonyms from the credential manager and the parameters from the key manager. Some operations are delegated to the smart card manager, which is described in Section 3.3. The Idemix library stores all the data related to the issuer, provers and verifiers in software i.e. in the credential an key mangers. The proof engine constructs the overall zero-knowledge proof based on the Fiat-Shamir heuristic [11]. Then, the ZkDirector class is used for generating the proof through the list of ZkModules created by the proof assembler. That class is based on the *builder* design pattern [13]. This pattern encapsulates the construction of an object step by step by telling a builder class how to create a certain class according to a set of options.

The context of the combined is proof is generated via the union of the contexts of the partial zero-knowledge proofs. This undermines the possibility of integrating IRMA as is now (Section 2). During the generation of the partial proofs different methods are called on each involved ZkModule . We are mainly interested in those that can be delegated to a hardware token:

– `firstRound`: it generates the *t_values* for each attribute and those that will be sent to the verifier. Moreover it prepares the values that both prover and verifier know (common values).

– `secondRound`: it generates the *s_values* through the generated challenge. Hence, this method undermines the possibility of generating the challenge by the card.

3.2 The CL building block

This building block produces the correspondent ZkModules when an operation related to a credential issued by the CL ABC system is required by a presentation token description. It generates a ZkModule identified by `sig:0` and initialized with the system and verifier parameters.

From here, certain cryptographic operations can be delegated to a hardware token:

– The computation of domain pseudonyms.
– The generation of the commitment related to the key binding use case (Section 4).
– The computation of a public key commitment.
– The generation of two *s-values* related to the key binding use case.

The Idemix library supports a hardware token with a certain structure withing the key binding use case. This hardware token should comprise a master secret (namely, the device secret, ms_0). Moreover, an additional master secret exists and is related to the credential that is being used in the proof. That master secret is considered the credential secret (ms_1).

When a credential based in the CL ABC system is involved in Idemix the signature triple (A, e, v) is first randomized:

$$r_A \in_R \{0,1\}^{l_n + l_\o} \tag{1}$$

$$A' = AS^{r_A} \mod n \tag{2}$$

$$v' = v - er_A \tag{3}$$

$$e' = e - 2^{l_e - 1} \tag{4}$$

Then, the card smart card computes the commitment T as:

$$T = R_s^{\tilde{ms}_0} R_d^{\tilde{ms}_1} \mod n \tag{5}$$

where $\tilde{ms}_0, \tilde{ms}_1$ are the pseudorandom values associated to the device secret and the credential secret respectively. The bases R_s, R_d are part of the public key of the issuer (Section 2). This commitment is part of the proof of knowledge that guarantees that the card knows both the device secret and the credential secret:

$$\mathsf{NIZK} : \{(ms_0, ms_1) : T = R_s^{ms_0} R_d^{ms_1} \mod n\} \tag{6}$$

The commitment T is retrieved and then inserted by the ZkModule for computing the following partial proof:

$$\mathsf{NIZK} : \{(a_1, ms_0, a_2, a_3, a_4, a_5, ms_1) : T = A'^{a_1} R_d^{ms_1} R_1^{a_2} R_2^{a_3} R_3^{a_4} R_s^{a5+ms_0} \mod n\}(A', T) \quad (7)$$

where $a_1 ... a_5$ are the randomizers involved in the proof.
Moreover, the card must generate two *s_values* for retrieving each ms_i:

$$s_1 = \tilde{ms}_0 + c \cdot ms_0 \qquad s_2 = \tilde{ms}_1 + c \cdot ms_1 \quad (8)$$

In the next section, we describe the main methods of the external secret manager in order to identify the locations where the integration takes place.

3.3 The external secret manager

This class interfaces a CL ZkModule with secret manager that can be a smart card. It generates randomizers, commitments, pseudonyms and response values whose operation can be delegated to a card. Besides, the external secret manager implements methods for saving and fetching issue public key and the system parameters (Table 1). These methods can be extended in certain cases for integration hardware tokens (e.g. in the alternative **c**, Section 3.4).

Table 1: Methods for fetching and updating issue and system parameters

Method	Description
`readSystemParametersIfNeeded`	Fetches the issuer public key
`readIssuerParametersIfNeeded`	Fetches the system parameters
`setSystemParameters`	Updates and sets the system parameters

The secret manager also describes methods for registering U-Prove and CL issuers via the `registerUProveIssuer` and `registerClIssuer` methods. The generation of the hardware token and credential secrets is performed in the `allocateCredential` method. At that point, we can intercept the secret values and uploaded into the smart card. Finally, the external secret manager contains individual methods for obtaining different values from the system parameters and issuer public key values:

All functions from Table 2 rely on the `readSystemParametersIfNeeded` and `readIssuerParametersIfNeeded` methods. Moreover, these methods could be used to communicate directly with the smart card in the case these values were stored there. In Table 3 we summarize the set of methods that can be used for delegating different cryptographic operations to the smart card:

Based on the functionality of these classes we provide three alternatives for integrating hardware tokens in the next section.

Table 2: Methods for fetching the cryptographic system parameters of a setup

Method	Description
`getPublicKeyBase`	Base utilized for performing public key commitments
`getPseudonymModulus`	Modulus utilized for computing exclusive scope pseudonyms
`getPseudonymSubgroupOrder`	Order of the subgroup utilized in the computation of exclusive scope pseudonyms
`getBaseForDeviceSecret`	Base utilized in the computation of the T commitment (Eq. 5)
`getBaseForCredentialSecret`	Bases utilized in the computation of the T commitment (Eq. 5)
`getModulus`	Modulus involved in the computation of commitments and proofs
`getChallengeSizeBytes`	Size of the challenge
`getRandomizerSizeBytes`	Size of the randomizers utilized in proofs
`getAttributeSizeBytes`	Size of the credential attributes

Table 3: Methods for performing cryptographic operations via the smart card manager

Method	Description
`generateRValues`	Generates the randomizers related to the device and credential secrets i.e. $\tilde{ms}_0$ and $\tilde{ms}_1$
`getCommitmentForCredential`	Generates the T commitment described in Eq. 6
`getCommitmentForScopeExclusivePseudonym`	Generates a domain pseudonym commitment
`getCommitmentForPublicKey`	Generates public key commitments
`getResponse`	Given a challenge c, computes a *s_value*. It implements the `getResponseForCredentialSecretKey` and `getResponseForDEviceSecretKey` methods for each *s_value* of Eq. 8

3.4 Integration alternatives

For every option we could design in the integration process, we must consider if the following operations are delegated or not to the card:

- **Issuing**: This comprises the issuing of credentials using the CL ABC or U-Prove
- **Loading of system parameters and the public key of involved issuers**: Those are required in the issuing and proving operations performed by the card. Moreover, one can consider the generation of the device and credential secrets by the card
- **Pseudonyms and public key commitments**: Defined by the CL ABC as randomized commitments to the master secret. The public key commitments are commitments to the device secret
- **Generation of *t_values***: This can be related to T (Eq. 5) or to the full proof (Eq. 7). This also comprises the generation of the randomizers for those values that are hidden and whose ownership is proved
- **Generation of *s_values***: This can be related to the generation of the *s_values* of Eq. 9 as well as the generation of the *s_values* for each partial proof, hidden attributes and properties of the attributes

We have identified three different alternatives for integrating the IRMA card:

1. **Alternative a**: key binding use case: This alternative is related to the procedure described by the Eqs. 6–9 in Section 3.3. Moreover, the computation

of limited scope pseudonyms and public key commitments are also supported by the card. Besides, the card must be stored the system parameters and the public key parameters in order to compute the T values. Finally, the card computes two *s_values* when the overall challenge is provided. As noted elsewhere, the generation of the challenge is centralized and that undermines the generation of it by the card, since it depends on the results of the partial proofs. We cannot rely on the IRMA card for generating it (as described in Section 2.1). Therefore, the IRMA card must be adapted for generating *s_values* i.e. $\hat{s} = \tilde{s} + c \cdot s$ for a certain value s where $\tilde{s}$ is a pseudorandom value, $\hat{s}$ is a hidden identifier and c is received.

2. **Alternative b**: a solution only based on smart cards. Since the cryptographic engine of the library can be easily replaced by injection, a solution solely based on smart cards can fit in the Idemix architecture. Given that the new library supports U-Prove and an open-source implementation for MULTOS cards already exists [15], it can be possible to design a card based on two applications: one for CL and another one for U-Prove and enable message passing in the card for generating a common challenge. Nonetheless, the limitations of this approach are related to the processing, RAM and storage capabilities of the card.
3. **Alternative c**: an intermediate approach. Several functionalities of the IRMA card (Section 2) can be still reused in the integration such as the randomization of the (A, e, v) tuple of the CL signature as well as the computation of Eq. 8. Besides, it must be possible to integrate the issuing operation of the CL ABC in the smart card. In order to avoid dealing with negative numbers in the card, we have increased the length of the v component of the CL signature to 1,700. This is required for randomizing signatures on the card. Moreover, it relies on the methods depicted in Table 2 for storing and retrieving system parameters and issuer public keys.

3.5 Summary

Since the current architecture of the External Secret Manager requires structural changes for implementing the operations described in the alternative **c** (particularly, the issuing operation) and the alternative **b** cannot withstand with the performance of nowadays smart cards (for instance, for performing range proofs or verifiable encryption), we rely on the first alternative in this manuscript.

Finally, we support exclusive scope pseudonyms and public key commitments on the card via additional APDUs. Finally, as noted in Section 2, we perform all the operations using 1,024 bit keys.

4 Integration of the key binding use case

In this section, we describe the required changes in the current Idemix library for performing the key binding use case in the IRMA card.

4.1 Software changes

We have implemented the communication with the IRMA card and interface its functionalities via two classes: `Irma` and `IrmaLoader`[8]. The first class, `Irma`, includes the definition of all the APDUs needed for uploading the required parameters in the card retrieving the results of the cryptographic operations. The following methods are exposed:

Table 4: Methods for uploading signatures, issuer public keys and system parameters into the smart card

Method	Description
`void uploadClSignature(void)`	Uploads a tuple (A, e, v) to the smart card of a certain credential.
`void uploadClPublicKey(void)`	Uploads the modulus and the bases for computing the T commitment defined in Eq. 5.
`void uploadSystemParameters(void)`	Uploads the modulus, base and generator for performing domain pseudonyms and public key commitments in the smart card.

Besides, this class implements methods for connecting to the smart card and selecting the IRMA application:

Table 5: Methods for managing IRMA cards

Method	Description
`List<CardTerminal> getTerminalList()`	Obtains a list of the available readers connected
`CardChannel getChannel(CardTerminal terminal)`	performs the connection to a given terminal
`boolean connectToIrma(CardChannel channel)`	Selects the IRMA application via its Application Identifier (AID)

Those functions relies on the `javax.smartcardio` API.

Finally, the methods below are used for performing cryptographic operations and request its results via APDUs. These functions returns `BigInt` objects. They are a wrapper of the JAVA `BigInteger` class.

All the values the methods above use are parsed and extracted from XML files following the ABC4Trust scheme [5]. The `IrmaLoader` class is responsible for parsing three types of XML files: credential specifications, issuer public keys and system parameters. Moreover, it parses presentation policies in order to extract the scope, needed for derive exclusive scope pseudonyms. This class exposes the following methods for parsing these files:

[8] Available at `https://github.com/adelapie/irma_integration`, (accessed 11 August, 2014).

Table 6: Methods for performing cryptographic operations via IRMA card

Method	Description
`BigInt getNYM()`	Computes and retrieves a domain pseudonym based on the based uploaded via `uploadClPublicKey` and `uploadSystemParameters`
`BigInt getCOM()`	Computes the T commitment defined in Eq 5. and retrieves it
`BigInt getS1(BigInt challenge)`	Computes the first *s_value* from Eq. 8
`BigInt getS2(BigInt challenge)`	Computes the second *s_value* from Eq. 8
`BigInt getPKCOM()`	Computes a public key commitment
`BigInt getRANDOM()`	Generates the randomizers $\tilde{ms}_0$ and $\tilde{ms}_1$

Table 7: Methods for parsing system parameters, issuer public keys and presentation policies

Method	Description
`boolean processClSignature()`	Parses a certain credential specification and uploads a (A, e, v) tuple in memory
`boolean processClPublicKey()`	Parses an issuer public key and loads its content in memory
`boolean processSystemParameters()`	Parses and process the public system parameters of a certain set up
`boolean processPresentationPolicy()`	Parses a certain presentation policy and extract the scope, required for deriving exclusive scope pseudonyms

4.2 Hardware changes

We have added a new set of APDUs for loading the issuer public key and the system parameters in the card. However, the actual functionality of the card (issuing and proving) is still available. Moreover, we have added support for performing new functionalities (T commitments, pseudonyms, public key commitments and the computations of *s_values* via a given challenge).

The new variables stored in static memory (EEPROM) are represented in Table 8.

Table 8: New variables added for supporting the key binding use case

Variable	Description
`unsigned char m_challenge[SIZE_H]`	Stores the challenge received by the terminal computed by the Idemix library and used in the card for computing the *s_values* corresponding to the commitment T
`unsigned char secret[SIZE_SECRET_ABC]`	Credential secret
`unsigned char secret2[SIZE_SECRET_ABC]`	Device secret
`unsigned char xr1[SIZE_RAND]`	First randomizer of commitment T.
`unsigned char xr2[SIZE_RAND]`	Second randomizer of commitment T.
`unsigned char r_base_nym[SIZE_N]`	This is the base utilized for computing the pseudonym related commitment.
`unsigned char r_dhgen_1[SIZE_N]`	This is the generator utilized in the public key commitments
`unsigned char r_mod_nym[SIZE_N]`	This is the modulus utilized for computing the pseudonym related commitment.

Variable sizes depicted in Table 8 are consistent with those utilized in the Idemix library i.e. `SIZE_N` = 128 bytes, `SIZE_SECRET_ABC` = 10 bytes, `SIZE_H` = 32 bytes (SHA-256) and `SIZE_RAND` = 52 bytes. In total, the integration of the key binding use case requires `SIZE_H` + `SIZE_SECRET_ABC`·2+ `SIZE_RAND`·2+ `SIZE_N`·3 = 540 bytes.

Computing commitments and response values is performed through the arithmetic functions available in the IRMA card[9] with size modifications and parameters (for instance, for deriving *s_values*). The implementation of exclusive domain pseudonyms corresponds to [10].

A note on alternative c As noted elsewhere, the size of the v component when a credential is issued has been increased to 1,700 in order make compatible the randomization function of the IRMA implementation. This change must be done at
`com.ibm.zurich.idmx.buildingBlock.systemParameters`, in the `EcryptSystemParametersWrapper` class. The method that returns the size of v (`public int getL_v()`) now is forced to return the new size. We have also implemented independent methods for extracting system parameters and the components of the public key of a certain issuer stored in the card via: `getS()`, `getZ()`, `getA()`, `getE()`, `getV()`, `getRd()`, `getRs()`, `getR0()`, `getR1()`, `getR2()`, `getR3()`, `getR4()`, etc. All these methods return an object of type `BigInt`. The randomization of a CL tuple is performed via two methods: `BigInt randomizeA()` (computes Eq. 2) and `BigInt randomizeV()` (computes Eq. 3).

5 Performance figures

In this section, we describe the performance figures related to the operations described in Section 4. As IRMA does, we rely on the MULTOS ML3-80K-R1 cards. The full description of the APDUSs can be found in the declaration of the `Irma` class.

We consider the following set up. First, the cardholder has received through issuing the issuing protocol a CL credential with the following structure (related to the ABC4Trust scheme described in [5]):

```xml
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<ns2:Credential xmlns:ns2="http://abc4trust.eu/wp2/abcschemav1.0">
    <ns2:CredentialDescription RevokedByIssuer="false">
        <ns2:CredentialUID>device-cred-da1fb3f2-5010-4655-99b4-6900d36d7587</ns2:CredentialUID>
        <ns2:FriendlyCredentialName lang="en">University Credential</ns2:FriendlyCredentialName>
        <ns2:ImageReference>https://idm.cti.gr/...</ns2:ImageReference>
        <ns2:CredentialSpecificationUID>urn:patras:credspec:credUniv</ns2:CredentialSpecificationUID>
        <ns2:IssuerParametersUID>urn:patras:issuer:idemix</ns2:IssuerParametersUID>
        <ns2:SecretReference>secret</ns2:SecretReference>
        <ns2:Attribute>
            <ns2:AttributeUID>-515302633</ns2:AttributeUID>
            <ns2:AttributeDescription Type="urn:patras:credspec:credUniv:firstname" DataType="xs:string"
                Encoding="urn:abc4trust:1.0:encoding:string:sha-256">
                <ns2:FriendlyAttributeName lang="en">first name</ns2:FriendlyAttributeName>
            </ns2:AttributeDescription>
            <ns2:AttributeValue xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
            xmlns:xs="http://www.w3.org/2001/XMLSchema" xsi:type="xs:int">222</ns2:AttributeValue>
        </ns2:Attribute>
        <ns2:Attribute>
            <ns2:AttributeUID>1022843428</ns2:AttributeUID>
            <ns2:AttributeDescription Type="urn:patras:credspec:credUniv:lastname" DataType="xs:string"
```

[9] `https://github.com/credentials/irma_card` (accessed 11 August, 2014).

```
             Encoding="urn:abc4trust:1.0:encoding:string:sha-256">
                <ns2:FriendlyAttributeName lang="en">last name</ns2:FriendlyAttributeName>
            </ns2:AttributeDescription>
            <ns2:AttributeValue xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
            xmlns:xs="http://www.w3.org/2001/XMLSchema" xsi:type="xs:int">333</ns2:AttributeValue>
        </ns2:Attribute>
        <ns2:Attribute>
            <ns2:AttributeUID>780802239</ns2:AttributeUID>
            <ns2:AttributeDescription Type="urn:patras:credspec:credUniv:university" DataType="xs:string"
            Encoding="urn:abc4trust:1.0:encoding:string:sha-256">
                <ns2:FriendlyAttributeName lang="en">university name</ns2:FriendlyAttributeName>
            </ns2:AttributeDescription>
            <ns2:AttributeValue xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
            xmlns:xs="http://www.w3.org/2001/XMLSchema" xsi:type="xs:int">444</ns2:AttributeValue>
        </ns2:Attribute>
        <ns2:Attribute>
            <ns2:AttributeUID>202036118</ns2:AttributeUID>
            <ns2:AttributeDescription Type="urn:patras:credspec:credUniv:department" DataType="xs:string"
            Encoding="urn:abc4trust:1.0:encoding:string:sha-256">
                <ns2:FriendlyAttributeName lang="en">department name</ns2:FriendlyAttributeName>
            </ns2:AttributeDescription>
            <ns2:AttributeValue xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
            xmlns:xs="http://www.w3.org/2001/XMLSchema" xsi:type="xs:int">666</ns2:AttributeValue>
        </ns2:Attribute>
        <ns2:Attribute>
            <ns2:AttributeUID>1591524571</ns2:AttributeUID>
            <ns2:AttributeDescription Type="urn:patras:credspec:credUniv:matriculationnr" DataType="xs:integer"
            Encoding="urn:abc4trust:1.0:encoding:integer:unsigned">
                <ns2:FriendlyAttributeName lang="en">matriculation number</ns2:FriendlyAttributeName>
            </ns2:AttributeDescription>
            <ns2:AttributeValue xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
            xmlns:xs="http://www.w3.org/2001/XMLSchema" xsi:type="xs:int">555</ns2:AttributeValue>
        </ns2:Attribute>
    </ns2:CredentialDescription>
    <ns2:CryptoParams>
        <ns2:Signature>
            <ns2:canReuseToken>true</ns2:canReuseToken>
            <ns2:SignatureToken>
                <ns2:Parameter xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                xsi:type="ns2:BigIntegerParameter" Name="A">
                    <ns2:Value>23682103363720665521158257186426375619097243955204122101756938511
                    2271694219081314552478901178573792670918385682521966704244624520792443831029
                    4372174546710649192111884768151043146380698409986097437974102998617800435627
                    6599943285608892602306677601442807747824082841973909896181380875135342280991
                    694603552190079</ns2:Value>
                </ns2:Parameter>
                <ns2:Parameter xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                xsi:type="ns2:BigIntegerParameter" Name="e">
                    <ns2:Value>25934472305506205990702549148069757193827788951515230624972858310
                    5665800713306759149981690559193987143046241049547203654793910123242547694870
                    4509063070505204050561108938740582864 31</ns2:Value>
                </ns2:Parameter>
                <ns2:Parameter xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                xsi:type="ns2:BigIntegerParameter" Name="v">
                    <ns2:Value>34272954133610780751730790855491178323855677448531016247298379445
                    6869076297518193684915606229306138196118542525186150204190184281499387180945
                    4258554708089816034071591326976341860077335842139662111962674832898429524322
                    5769945816371198682096402810080909250401634872747986508578958460371654451665
                    5965187252684333069161514140585181167564412613275767085415264238264324246956
                    4720533502590423681363915482550242307084966302037527279554509138160957920815
                    2989551252927314003438944028539181144236818331129131497675564204473</ns2:Value>
                </ns2:Parameter>
            </ns2:SignatureToken>
        </ns2:Signature>
    </ns2:CryptoParams>
</ns2:Credential>
```

In this example, the prover receives a presentation policy that requires to prove the ownership over that credential together with the utilization of a domain exclusive pseudonym:

```xml
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<abc:PresentationToken
  xmlns:abc="http://abc4trust.eu/wp2/abcschemav1.0"
  Version="Version 1.0"
  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:PresentationTokenDescription
        PolicyUID="urn:patras:policies:courseEvaluation"
        TokenUID="abc4trust.eu/token-uid/3">
    <abc:Message>
      <abc:Nonce>bkQydHBQWDR4TUZzbXJKYUphdVM=</abc:Nonce>
    </abc:Message>
    <abc:Pseudonym Exclusive="true" Scope="urn:patras:evaluation"
    SameKeyBindingAs="#credCourse"/>
    <abc:Credential Alias="#credUniv" SameKeyBindingAs="#credCourse">
        <abc:CredentialSpecUID>urn:patras:credspec:credUniv</abc:CredentialSpecUID>
        <abc:IssuerParametersUID>urn:patras:issuer:idemix</abc:IssuerParametersUID>
    </abc:Credential>
  </abc:PresentationTokenDescription>
</abc:PresentationToken>
```

First, the card must perform the computation of the T commitment (Eq. 5), generate the two *s_values* from Eq. 8 and compute an exclusive scope pseudonym. In the next section we evaluate the required time for computing those operations together with parsing and loading the public system parameters and an issuer public key into the card.

5.1 Results

We have depicted in Table 10 the performance of the new instruction added to the IRMA card for supporting the key binding use case of the Idemix library. The first group of instructions (`INS_SET_BASE`, `INS_SET_IPK` and `INS_SET_SIG`) are related to loading a certain issuer public key and credential signature. Then, we show the performance of all the required cryptographic operations described in Eqs. 5, 6 and 8.

According to Table 9 is possible to load all the required parameters for the key binding use case is 283.14 ms (one issuer public key, a CL tuple and the system parameters needed for computing exclusive scope pseudonyms). The time required (on card) for computing all the cryptographic operations involved in the presentation policy depicted in Section 5 is equivalent to 206.72 ms.

Table 9: Performance of on-card cryptographic operations

APDU	Description	P1	Parameter	Time (ms)
`INS_SET_BASE`	Uploads a certain base R_i into the card	0x00	R_0	19.581
		0x01	R_1	20.252
		0x02	R_2	20.175
		0x03	R_3	20.315
		0x04	R_4	20.339
		0x05	R_5	19.299
`INS_SET_IPK`	Uploads the public key parameters of an issuer into the card	0x00	R_t	20.123
		0x01	R_d	20.540
		0x02	mod	20.359
		0x03	Z	20.370
		0x04	S	20.249
`INS_SET_SIG`	Uploads a tuple (A, e, v) in the card	0x00	A	19.650
		0x01	e	18.996
		0x02	v	22.895
`INS_GET_NYM`	Computes an exclusive scope pseudonym	0x00	-	55.199
`INS_GET_COM`	Computes the commitment T (Eq. 5)	0x00	-	98.950
`INS_GET_S_1`	Computes the first *s_value* (Eq. 8)	0x00	-	19.112
`INS_GET_S_2`	Computes the second *s_value* (Eq. 8)	0x00	-	19.112
`INS_GET_PK`	Computes a public key commitment	0x00	-	54.235
`INS_GEN_RANDOM`	Generates two randomizers for T (Eq. 5)	0x00	-	14.352

6 Discussion

In the last few years different implementations of ABCs based on smart cards have been proposed. Bichsel et al. presented the first implementation of Idemix on the Java Card platform in 2009. Proving the possession of one credential with one attribute (i.e. the master secret), required 7.4 seconds (1,280-bit modulus) and 10.55 seconds with a modulus of 1,536 bits [2]. Sterckx et al. followed a similar approach for implementing the signing protocol of Direct Anonymous Attestation (DAA) in Java Cards [16]. In their design, one transaction requires 4.2 seconds using a modulus of 1,024 bits. Nonetheless, the Java Cards 2 that Bichsel et al. and Sterckx et al. relied on do not provide direct access to modular arithmetic operations. Consequently, these authors had to rely on different strategies for performing modular multiplications and exponentiations, thus undermining the overall performance of the implementation. The IRMA card is based on the MULTOS platform embedded on the Infineon SLE78 chip. In contrast, this chip supports a variety of modular arithmetic operations. Using credentials of 5 attributes, the disclosure of all the attributes requires 0.947 seconds whereas the worst case (hiding the 5 attributes) is performed in 1.454 seconds [17].

When revealing all the attributes but the master secret the IRMA card first randomizes the signature, computes the the commitment related to the components of the signature not revealed and the master secret and prepares the correspondent response values. In the key binding use case, we only need to perform two modular exponentiations in the commitment T, an extra modular exponentiation for the pseudonym commitment and then prepare two response values. Given the reduced number of modular arithmetic operations we only require 206.75 ms. We can expect a similar performance in other smart cards relying on the SLE78 chip.

References

1. Lejla Batina, Jaap-Henk Hoepman, Bart Jacobs, Wojciech Mostowski, and Pim Vullers. Developing efficient blinded attribute certificates on smart cards via pairings. In *CARDIS*, pages 209–222, 2010.
2. Patrik Bichsel, Jan Camenisch, Thomas Groß, and Victor Shoup. Anonymous credentials on a standard Java Card. In *ACM Conference on Computer and Communications Security*, pages 600–610, 2009.
3. Stefan A. Brands. *Rethinking Public Key Infrastructures and Digital Certificates: Building in Privacy.* MIT Press, Cambridge, MA, USA, 2000.
4. Jan Camenisch, Rafik Chaabouni, and Abhi Shelat. Efficient protocols for set membership and range proofs. In Josef Pieprzyk, editor, *ASIACRYPT*, volume 5350 of *Lecture Notes in Computer Science*, pages 234–252. Springer, 2008.
5. Jan Camenisch, Maria Dubovitskaya, Anja Lehmann, Gregory Neven, Christian Paquin, and Franz-Stefan Preiss. Concepts and languages for privacy-preserving attribute-based authentication. In Simone Fischer-Hübner, Elisabeth de Leeuw, and Chris Mitchell, editors, *IDMAN*, volume 396 of *IFIP Advances in Information and Communication Technology*, pages 34–52. Springer, 2013.
6. Jan Camenisch and Els Van Herreweghen. Design and implementation of the *idemix* anonymous credential system. In *ACM Conference on Computer and Communications Security*, pages 21–30, 2002.
7. Jan Camenisch and Anna Lysyanskaya. An efficient system for non-transferable anonymous credentials with optional anonymity revocation. In *Proceedings of the International Conference on the Theory and Application of Cryptographic Techniques: Advances in Cryptology*, EUROCRYPT '01, pages 93–118, London, UK, UK, 2001. Springer-Verlag.
8. Jan Camenisch and Anna Lysyanskaya. A signature scheme with efficient protocols. In *Proceedings of the 3rd international conference on Security in communication networks*, SCN'02, pages 268–289, Berlin, Heidelberg, 2003. Springer-Verlag.
9. Jan Camenisch and Victor Shoup. Practical verifiable encryption and decryption of discrete logarithms. In *CRYPTO*, pages 126–144, 2003.
10. Antonio de la Piedra, Jaap-Henk Hoepman, and Pim Vullers. Towards a full-featured implementation of attribute based credentials on smart cards. Cryptology ePrint Archive, Report 2014/684, 2014. `http://eprint.iacr.org/`.
11. Amos Fiat and Adi Shamir. How to prove yourself: Practical solutions to identification and signature problems. In *CRYPTO*, pages 186–194, 1986.
12. Eiichiro Fujisaki and Tatsuaki Okamoto. Statistical zero knowledge protocols to prove modular polynomial relations. In *CRYPTO*, pages 16–30, 1997.
13. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.
14. ISO/IEC. International standard 7816–4.
15. Wojciech Mostowski and Pim Vullers. Efficient U-Prove implementation for anonymous credentials on smart cards. In Muttukrishnan Rajarajan, Fred Piper, Haining Wang, and George Kesidis, editors, *SecureComm*, volume 96 of *Lecture Notes of*

the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering, pages 243–260. Springer, 2011.

16. Michael Sterckx, Benedikt Gierlichs, Bart Preneel, and Ingrid Verbauwhede. Efficient implementation of anonymous credentials on java card smart cards. In *1st IEEE International Workshop on Information Forensics and Security (WIFS 2009)*, pages 106–110, London,UK, 2009. IEEE.
17. Pim Vullers and Gergely Alpár. Efficient selective disclosure on smart cards using Idemix. In Simone Fischer-H editor, *3rd IFIP WG 11.6 Working Conference on Policies and Research in Identity Management, IDMAN 2013, London, UK, April 8-9, 2013. Proceedings.*