Advanced Applications for e-ID Cards in Flanders

ADAPID Deliverable D15

E-Government II

C. Troncoso (Ed.), C. Diaz, V. Gornishki, V. Sucasas, M. Kohlweiss, M. Sterckx and J. Balasch

January 2010
Executive Summary

This report presents the work done within ADAPID on advanced technologies for e-ID cards. Current e-ID cards are based on X.509 certificates, which provide authentication and digital signature functionalities. The shortcomings of X.509 certificates are that all data in the certificates is disclosed in every transaction, and that all transactions made with the same card are linkable to each other.

We have designed and implemented an advanced proof-of-concept smart card prototype that demonstrates the feasibility of implementing anonymous credential protocols in a smart card. Anonymous credential protocols allow for secure and privacy-enhanced authentication. The main advantages of these protocols over traditional PKI-based technologies is that they allow for data minimization and unlinkability. This implies that in each transaction the protocols only disclose the information that is strictly necessary for the provision of the service.

In order to achieve good performance, our implementation splits operations between the smart card and a client PC. Operations that require the secret keys are performed by the card, while other operations are outsourced to the PC to accelerate the performance of the system.

Our case-study application is an electronic petition service. Our implementation simultaneously provides the following properties: (i) petition signing is anonymous; (2) each citizen can only sign a petition once; and (3) signatures of different petitions by the same citizen are unlinkable to each other.
Contents

1 Introduction 9
  1.1 Case study: electronic petitions 10
  1.2 Contributions 11
  1.3 Roadmap deliverable 11

I Theory and Preliminaries 13

2 Cryptographic Preliminaries 15
  2.1 Notation 15
  2.2 Abstract Algebra 16
  2.3 Number Theory Problems 18
  2.4 Commitment Schemes 18
  2.5 Zero-Knowledge Proofs 20
    2.5.1 Zero-knowledge proofs of knowledge about discrete logarithms 20
  2.6 Camenisch-Lysyanskaya (CL) Signature 24

3 The DAA Protocol 27
  3.1 Introduction 27
  3.2 High-Level Description 29
  3.3 Security parameters 32
  3.4 Issuer’s Public Key 33
  3.5 Join Protocol 33
  3.6 Signing protocol 36

4 Smart Cards and the Java Card Standard 41
  4.1 Smart Card Basics 41
    4.1.1 Hardware 42
    4.1.2 Communication Model 44
  4.2 Java Card 44
    4.2.1 Java Language Subset 45
    4.2.2 Java Card Converter 45
II Design

5 Designing an Anonymous Credential System

5.1 Simplifications
5.1.1 Pseudonyms
5.1.2 The secret value f
5.1.3 Security parameters
5.1.4 Changes in the signature of knowledge
5.1.5 Refactoring

5.2 Final Design
5.2.1 Key Generation
5.2.2 Join Protocol
5.2.3 Signing Protocol

6 Privacy Preserving e-Petitions

6.1 Architecture
6.1.1 Issuer component
6.1.2 Verifier component
6.1.3 Client component

6.2 Architecture Requirements
6.2.1 Security
6.2.2 Privacy
6.2.3 Usability

III Implementation

7 e-Petition System Implementation

7.1 Implementation
7.1.1 Language choice
7.1.2 Application layers

8 Software Implementation

8.1 Program Design
8.1.1 Class PKI
8.1.2 Class certificate
8.1.3 Class Proof
8.1.4 Class Prover
8.1.5 Class CARD
8.1.6 Class SERVER
8.1.7 Class ISSUER

8.2 Code Specification
CONTENTS

8.2.1 List of functions .............................................. 85

9 Smart Card Implementation ....................................... 89
  9.1 Overview ....................................................... 89
    9.1.1 Functionality Description ................................ 90
    9.1.2 Problems .................................................. 90
    9.1.3 Development Tools ......................................... 91
    9.1.4 Platforms .................................................. 91
  9.2 Large Number Addition and Subtraction ....................... 92
  9.3 Modular Multi-Exponentiation ................................. 93
    9.3.1 RSA and Exponentiation .................................. 94
    9.3.2 RSA and Java Card ........................................ 94
    9.3.3 Modular Multi-Exponentiation ............................. 97
  9.4 Large Number Multiplication ................................ 98
    9.4.1 Large Integer Multiplication ............................. 98
    9.4.2 Modular Multiplication ................................... 99
    9.4.3 Code Optimisation ....................................... 102
    9.4.4 Memory Management ..................................... 102
  9.5 Results ....................................................... 102
    9.5.1 Timing Details .......................................... 102
    9.5.2 Memory Consumption .................................... 103
  9.6 Implementation Details ...................................... 103
    9.6.1 List of Functions ....................................... 103
    9.6.2 APDU Commands ......................................... 106

10 Conclusions ..................................................... 107
Chapter 1

Introduction

ADAPID is a project with a strong focus on security and privacy for e-ID card applications and technologies. In the first two years of the project, the goal was to develop secure and privacy-enhanced services compatible with deployed Belgian e-IDs, and our work in the area of e-government applications was presented in [22]. In the last two years, we have explored new technologies for developing advanced e-IDs, and we describe the results of our work in this report.

Currently deployed e-ID cards, based on X.509 certificates, provide authentication and digital signature functionalities to citizens, that allow them to access a variety of services. X.509 certificates have however certain limitations. First, all information contained in the certificate is made available for all transactions, even if some of this information is not necessary for the provision of some services. This implies that the data minimization requirements of the European Data Protection Directive are not fulfilled. Second, all transactions signed by the e-ID card contain unique identifiers, meaning that they are all linkable. As a result of these limitations, X.509 certificates are not suitable for implementing services in a privacy-preserving manner.

Our research has focused on advanced cryptographic techniques that generalize the functionality of X.509 certificates while addressing their limitations. For this purpose, we have designed and implemented a card prototype based on anonymous credentials. Anonymous credentials enable secure and privacy-enhanced identity management, and allow for the minimization of the amount of information disclosed in each transaction, as well as supporting unlinkability of transactions made with the same card.

We have chosen electronic petitions as case study for our prototype implementation. The next section motivates the choice of this application and provides some background on electronic petitions. Section 1.2 outlines our main contributions, and we briefly provide a roadmap for this report in Section 1.3.
1.1 Case study: electronic petitions

A petition is a formal request addressed to an authority and signed by numerous individuals. Through petitions, citizens are able to express their support or dissatisfaction with government initiatives, and provide feedback to government institutions. In the physical world, petition signers typically provide a unique identifier (such as the national ID number) together with their handwritten signature, so that fake or duplicate signatures can be eliminated.

Given the high cost of collecting and verifying petition signatures by hand, it is not surprising that petitions are increasingly available online. E-petitions present substantial advantages with respect to physical world petitions: it is much easier to reach a large number of people potentially interested on signing them, and the signature verification process can be automated. But they also introduce new security and privacy challenges.

Many of the currently available electronic petitions simply collect the name and national ID number of signers. Given that this information is not secret, it is impossible to check that the petition signer is really providing her own data. In other words, it is not possible to detect cheating, which diminishes the trustworthiness of the petition signature list. To prevent this, some e-petition servers check the IP address of the signer and allow only one signature per IP address. But this disenfranchises legitimate signers who share the IP address with other people (note that in some organizations thousands of users share the same IP address).

To ensure that an electronic petition signature is unique and legitimate, it is necessary to use cryptographic means such as digital signatures. Assuming that citizens possess electronic e-ID cards (as is the case in Belgium), an obvious way to implement e-petitions is to have citizens sign them using the key pair available on their e-ID card. However, such a solution is problematic from a privacy point of view. The e-ID public key certificate (needed to verify the digital signature) contains a lot of information about the holder of the card, such as her name, National Registry Number, and date of birth. Revealing all this information for the purposes of signing a petition would definitely be against the data minimization principle, which is the legal philosophy underpinning data protection regulation. Data minimization constitutes that minimal amounts of personal data may be processed, but only in as far as strictly necessary for legitimate purposes. In other words, processing of data must be adequate, relevant and not excessive in relation to the purposes of collection and processing.

While signer identification is required in the physical world to ensure the uniqueness and validity of signatures, it is possible to reconcile functional, security and privacy requirements in its electronic version using cryptographic techniques. We propose using anonymous credential protocols to ensure that (i) signatures are legitimate, (ii) each citizen can sign a petition at most once, and (iii) petition signers are anonymous.
1.2 Contributions

In this work we use the DAA protocol to build a credential system with smart card support. Therefore, we focus on implementing a simplified protocol that allows users to authenticate to a verifier, while holding the anonymity property and preventing the abuse of credentials.

Anonymous credential protocols have long been considered too expensive in terms of computational requirements to be implemented in resource-constrained devices such as smart cards. In order to achieve a good performance, we split the protocol operations between the card and a client PC. The card contains the cryptographic secrets and performs all operations that take these secrets as input. Expensive operations that do not require using the secrets are outsourced to the client PC. In this way, we achieve good levels of performance while providing maximum security, as the cryptographic secrets never leave the smart card.

1.3 Roadmap deliverable

The rest of this report is organized in three parts. The first part provides the necessary background to understand this work. Chapter 2 gives an overview of the cryptographic primitives used by the Direct Anonymous Attestation Protocol presented in Chapter 3. We describe the smart card technology in Chapter 4. The second part outlines the design of the system. Chapter 5 describes an anonymous credential system with smart card support. An application using such a system is presented in Chapter 6. The last part, Chapters 7, 8, and 9, describe our implementation. Finally, we offer our conclusions in Chapter 10.
Part I

Theory and Preliminaries
Chapter 2

Cryptographic Preliminaries

In this chapter we present some of the mathematical background useful for understanding this report. We start by introducing the notation and some concepts of abstract algebra. Subsequently, some definitions and facts corresponding to number theory are described. The aim of these first sections is make the report self-contained. For a more detailed explanation on these topics, we refer the reader to [29]. From this point on, we present cryptographic techniques that make use of the problems stated in number theory. We start by presenting the commitment schemes and the zero-knowledge proofs. The Camenisch-Lysyanskaya signature scheme follows. We finish this chapter by explaining the concept of anonymous credential systems.

2.1 Notation

We define a set as a collection of objects that satisfy some property. To denote a set two options are available:

- Enumerate the elements of the set using the set notation, e.g., \{1, 2, \ldots, 50\}.
- Using the interval notation. In this case, we distinguish between closed intervals, e.g., \[1 \ldots 100\], and open intervals, e.g., \]0 \ldots 101[.

We write a set using capital letters, e.g., \(M = \{m_1, m_2, \ldots, m_n\}\). The number of elements of a set is denoted \(|M|\), and by \(m_1 \in M\) we indicate that the element \(m_1\) belongs to the set \(M\). Given two sets \(M\) and \(R\), \(M \subseteq R\) means that \(M\) is a subset of \(R\). By \(M \subset R\) we indicate that \(M\) is a proper subset of \(R\), i.e., \(M \subseteq R\) and \(M \neq R\).

Finally, we denote by \(\mathbb{N}\), \(\mathbb{Z}\) and \(\mathbb{R}\) the sets of non-negative integers, the set of integers and the set of real numbers, respectively.
2.2 Abstract Algebra

Abstract algebra is the subject of mathematics that studies algebraic structures, such as groups, rings, fields or vector spaces. In this section however, we are interested primarily in groups.

Definition 2.1 (Binary Operation on a Set). A binary operation $*$ on a set $S$, is a mapping $S \times S \rightarrow S$. In other words, it is a rule that assigns an element of $S$ to each ordered pair of elements from $S$. Important properties that a binary operation can fulfill are:

- **Associative**: For all $a, b, c \in S$, $(a * b) * c = a * (b * c)$.
- **Commutative**: For all $a, b \in S$, $a * b = b * a$.
- **Identity element**: An element $e$ is the identity element if for all $a \in S$, $a * e = e * a = 1$.
- **Inverse**: An element $b \in S$ is the inverse of an element $a \in S$, if $a * b = b * a = e$.

If we use additive notation, the identity element $e$ is 0 and the inverse element is denoted by $-a$. In the multiplicative notation however, the identity element $e$ is 1 and the inverse element is denoted by $a^{-1}$.

Definition 2.2 (Group). A group $(G, *)$ is a set $G$ with an associative binary operation on $G$, satisfying two conditions:

1. The set $G$ contains an identity element for $*$.
2. All the elements in $G$ contain an inverse element under $*$.

If the operation $*$ is also commutative, the group is called **abelian** or **commutative**. The number of elements of a group is called its order. If the order is finite, then the group is also called **finite**.

From this point on, we refer to the group $(G, *)$ as $G$, when there is no risk of ambiguity.

Definition 2.3 (Euler’s totient function). Given $n \geq 1$, the Euler’s totient function $\phi(n)$ counts the number of positive integers in the interval $[1, n]$ that are relatively prime to $n$. The general formula is given by:

$$\phi(n) = n \cdot \prod_{i=1}^{m} \left(1 - \frac{1}{p_i}\right),$$

where $n = p_1^{e_1} \cdots p_m^{e_m}$ with $p_1, \cdots, p_m$ the prime factors of $n$. 
Fact 2.4. Given a prime $p$, the Euler’s totient function is $\phi(p) = p - 1$.

The group we are interested in this work is the one formed by the integers modulo $n$. The next properties of the groups are defined over this set.

Definition 2.5 (Integers modulo $n$). We refer to the integers modulo $n$ as the set $\mathbb{Z}_n$ with elements $\{0, 1, 2, \ldots, n - 1\}$.

Definition 2.6 (Multiplicative inverse). Given an element $a \in \mathbb{Z}_n$, we call $x \in \mathbb{Z}_n$ the multiplicative inverse modulo $n$ of $a$ if $ax \equiv 1 \pmod{n}$. If such an $x$ exists then it is unique, and $a$ is called invertible. The inverse of $a$ is denoted by $a^{-1}$.

Fact 2.7. An element $a \in \mathbb{Z}_n$ is invertible if and only if $\gcd(a, n) = 1$.

Definition 2.8 (Multiplicative group). Given a set $\mathbb{Z}_n$, we let the multiplicative group $\mathbb{Z}_n^*$ be the set:

$$\mathbb{Z}_n^* = \{a \in \mathbb{Z}_n \mid \gcd(a, n) = 1\}.$$ 

Fact 2.9. If $n$ is prime, then the multiplicative group $\mathbb{Z}_n^*$ is:

$$\mathbb{Z}_n^* = \{a \mid 1 \leq a \leq (n - 1)\}.$$ 

Definition 2.10 (Order of a group). The order of $\mathbb{Z}_n^*$, represented by $|\mathbb{Z}_n^*|$, is the number of elements in $\mathbb{Z}_n^*$.

Fact 2.11. It follows from Def. 2.3 of the Euler’s totient function, that $|\mathbb{Z}_n^*| = \phi(n)$.

Definition 2.12 (Order of an element). Given a value $a \in \mathbb{Z}_n^*$, the order of $a$ is denoted by $\text{ord}(a)$, and is the least positive integer $t$ such that:

$$a^t \equiv 1 \pmod{n}.$$ 

Definition 2.13 (Cyclic group). Given a value $\alpha \in \mathbb{Z}_n^*$ with order $\text{ord}(\alpha) = \phi(n)$, then $\alpha$ is a generator of $\mathbb{Z}_n^*$. If $\mathbb{Z}_n^*$ has a generator, then $\mathbb{Z}_n^*$ it is called cyclic.

Definition 2.14 (Quadratic residue). Given an element $a \in \mathbb{Z}_n^*$, $a$ is a quadratic residue modulo $n$ if there exists an element $a \in \mathbb{Z}_n^*$ such that $x^2 \equiv a \pmod{n}$. If this element $a$ does not exist, then $x$ is called a non-quadratic residue modulo $n$.

The set of all quadratic residues modulo $n$ is denoted by $QR_n$, while the set of all non-quadratic residues modulo $n$ is denoted by $\overline{QR}_n$. 

2.2. ABSTRACT ALGEBRA

17
CHAPTER 2. CRYPTOGRAPHIC PRELIMINARIES

2.3 Number Theory Problems

In this section we introduce some concepts that ensure the security of the constructions in the report. This security is based on the intractability of the problems defined.

**Definition 2.15 (Discrete Logarithm Problem).** Let $G$ be a cyclic group of order $q$ and $g$ a generator of $G$. Given a random value $x$, the discrete logarithm problem states that it is difficult to find the value $x$ such that:

$$g^x = h.$$ 

**Definition 2.16 (RSA Modulus).** An RSA modulus $n$ is a $2k$-bit number resulting of the product of two $k$-bit prime numbers $p$ and $q$, such that $n = p \cdot q$.

**Fact 2.17.** If $n = p \cdot q$ is an RSA modulus, then the Euler totient function is $\phi(n) = (p - 1)(q - 1)$.

**Definition 2.18 (RSA Assumption).** The RSA assumption states that, given a positive integer $e > 1$, an RSA modulus $n$ and a random element $u \in \mathbb{Z}_n^*$, it is hard to compute the value $v \in \mathbb{Z}_n^*$ such that:

$$v^e \equiv u \pmod{n}.$$ 

**Definition 2.19 (Safe Prime).** A prime number $p$ is called safe if $p = 2p' + 1$, where $p'$ is also a prime number. The number $p'$ is known as a Sophie Germain prime.

**Definition 2.20 (Special RSA Modulus).** An RSA modulus $n = p \cdot q$ is special if both $p$ and $q$ are safe primes, i.e., if $n = (2p' + 1)(2q' + 1)$.

**Fact 2.21.** If $n$ is a special RSA modulus, then $QR_n$ is a cyclic group under multiplication, of size $p'q'$, where all but $p' + q'$ of the elements are generators.

**Definition 2.22 (Strong RSA Assumption).** The strong RSA assumption states that, given an RSA modulus $n$ and a random element $u \in \mathbb{Z}_n^*$, it is hard to compute the values $e > 1$ and $v \in \mathbb{Z}_n^*$ such that:

$$v^e \equiv u \pmod{n}.$$ 

2.4 Commitment Schemes

A commitment scheme allows an entity to commit to a value while keeping it secret. After building the commitment, the entity can prove that it committed
to a specific value by opening the commitment. Usually this is done by revealing the secret randomness with which the commitment was built. A commitment can be seen as a sealed. Once the envelope is created, the secret value inside the envelope cannot be changed and nobody can open the envelope without the committer’s permission (with committer we refer to the entity making the commitment). Thus, as stated in [24] “Commitment schemes are the digital analogue to nontransparent sealed envelopes”.

In summary, two properties must be fulfilled by a commitment scheme. The first one is called hiding property and states that the committed value must be hidden. The second one is the binding property, and states that the committed value can not be changed later on by the committer. In other words, it must be impossible to find another value to obtain the same commitment.

For our implementation we use a commitment scheme based in groups of hidden order. This scheme was proposed by Damgård and Fujisaki [21] and is based on the group of quadratic residues modulo a special RSA modulus $n$, whose notation is $QR_n$. In this group almost all values are generators. In fact, we consider all values that belong to $QR_n$ as generators, because each element of such group only has negligible probability not to be a generator. This group, $QR_n$, satisfies the four criteria required by [21].

In this scheme we require a trusted third party that creates the public values $(n,R,S)$ such that $n$ is a special RSA modulus and the two values $R$ and $S$ belong to $QR_n$. This scheme requires the fulfillment of $R \in \langle S \rangle$. This is ensured by choosing $S$ and $R$ from $QR_n$, since both values are generators, i.e., $R \in \langle S \rangle$ and $S \in \langle R \rangle$ always hold.

Let $f$ be the value that the entity wants to commit to, and $v$ a value computed at random (and so, not related in any way to $f$). Then a commitment to the secret value $f$ and the random value $v$ is given by the expression:

$$\text{commit}_{(n,R,S)}(f,v) = R^f S^v \pmod{n}.$$ 

Note that, if we compute the commitment with the expression:

$$\text{commit}_{(n,R)}(f) = R^f \pmod{n},$$

the security of the scheme, in terms of the binding property and hiding property, holds under the RSA assumption. In such a way that is not possible to find a value $x$ such that $\text{commit}_{(n,R)}(f) = \text{commit}_{(n,R)}(x) = R^x \pmod{n}$ (binding property) and neither to retrieve the value $f$ without the prime factors of $n$ (hiding property). The inclusion of the exponentiation $S^v$ in this scheme is done to achieve the semantic security property.

A commitment scheme is said to be semantically secure if two commitments $\text{commit}(f_1)$ and $\text{commit}(f_2)$ are not distinguishable even in case of knowing the secret values $f_1$ and $f_2$. We can neither distinguish between two commitments on the same value.

There are other commitment schemes, as the one described by Pedersen [33], that are not based in the RSA assumption. In fact, the security in the Pedersen scheme is based on the assumption that computing discrete logarithms in a group of prime order is hard.
2.5 Zero-Knowledge Proofs

A zero-knowledge proof (concept introduced in [34]) is an interactive method for one party to prove to another that a statement is true, without revealing anything other than the veracity of the statement. These proofs merge two concepts, interactive proof and zero-knowledge.

An interactive proof is a two-party protocol between a prover $P$ and a verifier $V$. In this protocol $P$ proves to $V$ the veracity of some statement in such a way that if the statement is true the proof always succeeds. Otherwise, if some dishonest prover wants to prove a false statement there exists a probability that the proof does not succeed.

Formally speaking, an interactive proof for a language $L$ is a two party protocol where $P$ wants to prove $V$ that $x \in L$ holds. These entities follow a strategy that has the following properties:

Completeness: If $x \in L$ holds then $\Pr[P(x) \leftrightarrow V(x) \rightarrow \text{accept}] = 1$.

Soundness: If $x /\in L$ holds then $\Pr[\tilde{P}(x) \leftrightarrow V(x) \rightarrow \text{accept}] \leq \frac{1}{2}$.

Note that we can perform the proof many times to decrease the latter probability. After performing the proof $N$ times the probability is given by this expression $\Pr[\tilde{P}(x) \leftrightarrow V(x) \rightarrow \text{accept}] \leq \left(\frac{1}{2}\right)^N$.

The zero-knowledge property requires that the verifier cannot learn anything about the statement $x$ through performing the proof. Even in case of having a dishonest verifier $\tilde{V}$, this verifier only can learn the veracity of the statement.

2.5.1 Zero-knowledge proofs of knowledge about discrete logarithms

In our protocol we use signatures of knowledge based on groups of hidden order, but to understand this concept we first describe several proofs of knowledge based on the discrete logarithm problem.

First, we illustrate in Protocol 1 the Schnorr protocol [14]. In this proof of knowledge the prover $P$ has the knowledge of some secret value $x$, and the base $g$ is a public value. This value $g$ is a generator of the group $G_q$ of known order $q$. To prove the knowledge of the secret value, $P$ computes and publishes $y = g^x$, so $y$ is also public. This is denoted in short by $PK\{(x) : y = g^x\}$.

In a first step the prover $P$ commits to a random value $r$ by computing the value $t = g^r$. After receiving the commitment, the verifier $V$ picks a random challenge $c$. Using this challenge $c$, the committed value $r$ and the secret value $x$, the prover computes the response $s = r - cx \mod q$ such that $g^s y^c = t$.

We must take into account that a dishonest prover can succeed in this proof only if he can guess the challenge $c$ while computing the value $t$ in such a way that the prover can compute the value $t$ to succeed the verification without the knowledge of $x$. This probability becomes negligible if the range from which the challenge $c$ is chosen is sufficiently large.

In case of having a dishonest verifier, instead of choosing the challenge $c$ at random he computes a challenge depending on the received value $t$. This way,
2.5. ZERO-KNOWLEDGE PROOFS

Protocol 1 The Schnorr protocol

\[
\begin{array}{c|c|c}
\text{PK}\{(x) : y = g^x\} & \text{PROVER} & \text{VERIFIER} \\
\hline
(y, x) & (y) & \\
\downarrow & \downarrow & \\
\text{PROVER} & \text{VERIFIER} & \\
\hline
r \in \mathbb{Z}_q, t = g^r & t \rightarrow c \rightarrow c \in \{0, 1\}^k \\
s = r - cx \mod q & c \leftarrow s \rightarrow g^s y^c \equiv t
\end{array}
\]

the verifier can obtain some knowledge about the secret value \(x\). A suitable way of preventing this possibility is by precomputing the challenge. This means, that before receiving the value \(t\), the verifier chooses and commits to the challenge \(c\). This way the prover cannot know the value \(c\) but it can be sure that such value was computed before \(t\).

In our system we use proofs of knowledge based on groups of hidden order. We use the group of quadratic residues of a RSA modulus \(n\), i.e., \(QR_n\). If the modulus \(n\) is computed using the Sophie Germain primes \(p'\) and \(q'\) such that \(n = (2p' + 1)(2q' + 1)\), then the order of \(QR_n\) is \(p'q'\). Thus, finding the order of \(QR_n\) is as hard as finding the prime factors of \(n\). Without the knowledge of such prime factors we can only make the approximation for the order of \(QR_n\) with the expression \(\left\lfloor \frac{n}{4} \right\rfloor\). Another feature of this group is that almost all members are generators, in fact, the number of generators is given by the expression \(p'q' - p' - q'\), so the probability of finding a no-generator member is \(p = \frac{p'q' - p' - q'}{p'q'}\).

We must take into account that all operations, except the value \(s\), are done in \(QR_n\), and so all operations are operations modulo \(n\). Due to this, and having the public value \(g\) as a generator of \(QR_n\), the value \(y = g^x\) is also a generator of \(QR_n\). Protocol 2 illustrates this protocol.

The problem of this scheme is that the prover does not know the order of \(QR_n\) and so the response \(s\) cannot be computed following \(s = r - cx \mod p'q'\). The response is computed in \(\mathbb{Z}\), and so the verifier can obtain some information about the secret value. As an example, if the verifier chooses a large challenge \(c\) and the value \(s\) is still large, the verifier learns that \(x\) is rather small. To prevent this situation the random value \(r\) is chosen from a larger range than \(x\). The ranges used to ensure the security of the scheme are shown in Table 2.1. As can be see in the table, the ranges of the values \(r\) and \(s\) are very similar.

In summary, the main differences between proofs of knowledge based on groups of hidden order are that (1) the public value \(g\) and the public value \(y\) are generators of \(QR_n\), (2) the random value \(r\) is chosen from a long interval,


**Protocol 2** The Schnorr protocol based on the group $QR_n$.

$$
\begin{array}{c|c}
\text{PROVER} & \text{VERIFIER} \\
\hline
(y, x) & (y) \\
\hline
\downarrow & \downarrow \\
\text{PROVER} & \text{VERIFIER} \\
\hline
r \in \mathbb{Z}_q, t = g^r & c \in \mathbb{R}\{0,1\}^k \\
\hline
s = r - cx & g^s y^c = t \\
\end{array}
$$

Table 2.1: Ranges for the values in the Schnorr protocol based on $QR_n$.

<table>
<thead>
<tr>
<th>Value</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>secret value $x$</td>
<td>$[0 \ldots n2^k]$</td>
</tr>
<tr>
<td>challenge $c$</td>
<td>$[0 \ldots 2^k] = {0,1}^k$</td>
</tr>
<tr>
<td>random value $r$</td>
<td>$[-n2^{-2+k+k'} \ldots n2^{-2+k+k'}]$</td>
</tr>
<tr>
<td>response $s$</td>
<td>$[-n2^{-1+k+k'} \ldots n2^{-1+k+k'}]$</td>
</tr>
</tbody>
</table>

(3) the response $s$ is not a modular operation and so belongs to $\mathbb{Z}$ and (4) all the rest of computations are done in $QR_n$.

We have explained two different ways of proving the knowledge of one secret value $x$, but these protocols can be extended to prove the knowledge of a representation. If the secret we want to prove is the vector of values $x_1 \ldots x_l$, and we have a vector of public values with the same number of bases $g_1 \ldots g_l$, we can perform the protocol presented in Protocol 3. In this protocol the intervals from where the number are chosen are represented by capital letters. Note that the proof of a representation can be done either for a group of hidden order or known order.

This scheme can be extended to prove knowledge of several representations, i.e., with more than one public key. It can be seen as performing several proofs of a representation, but sharing the challenge $c$ [27]. We do not elaborate on this because in our protocol we use proofs of representation with only one public key. We convert this proof of knowledge, using the Fiat-Shamir Heuristic [23], into a non-interactive proof of knowledge called signature of knowledge. The only difference between the proofs of knowledge previously illustrated and the
Protocol 3 Proof of a representation.

\[ PK\{(x_1 \ldots x_l) : y = \prod_{i=1}^{l} g_i^{x_i}\} \]

\[ (y, x_1 \ldots x_l) \]

\[ \downarrow \]

\[ PROVER \]

\[ (y) \]

\[ \downarrow \]

\[ VERIFIER \]

\[ r_i \in_R R, t = \prod_{i=1}^{l} g_i^{r_i} \]

\[ t \]

\[ c \]

\[ c \in_R C \]

\[ s \]

\[ s = r_i - cx_i \in [1..l] \]

\[ (\prod_{i=1}^{l} g_i^{s_i}) y^c = t \]

signature of knowledge that we describe in our design is that the challenge \( c \) is computed by the prover. Of course, if the prover knows the challenge while computing the value \( t \) he can forge this proof. To prevent this situation, we compute the value \( c \) using a cryptographic hash function whose inputs are the commitments of the secret values we want to prove and all public parameters. Thus, after receiving the \( s \)-values (responses) and the commitment (value \( t \)), the verifier can check whether or not the challenge was correctly computed.

We also need interval checks in our signature of knowledge. Proofs of knowledge with interval checks allow an entity to prove not only the knowledge of a value, but also that such value lies in an interval \( \Gamma = [-2^l, 2^l] \). We explain an efficient scheme, described in [32], that is based on groups of hidden order. The principal idea of this scheme is that the prover has to choose the random value \( r \) from an interval \( R(1^l) \) in order to compute a response \( s = r - cx \) that lies in some interval \( S(1^l) \). The verifier, besides checking the verification \( g^s y^c = t \) also checks whether or not the value \( s \) lies in such interval. If the secret value \( x \) does not lie in the required interval the prover only has one chance to succeed the proof, that consists in choosing one exact value \( r \) that depends on the challenge \( c \). Thus, dishonest provers have to guess the challenge to forge the proof. This proof is shown in Protocol 4 and the intervals for the values are shown in Table 2.2.

The proof allows to check if the secret value \( x \) lies in an interval \( \Gamma = [-2^l, 2^l] \), this is used in the signature of knowledge we use in our protocol. However, for the implementation of some additional features we need proofs that \( x \) lies in an interval \( [a, b] \). Note that this proof is different from the one previously defined. The interval is not symmetric, i.e., the bounds \( a \) and \( b \) can have the same sign and \( x \) does not need to be chosen from a smaller interval \( [-2^l - k \cdots 2^l - k] \).

A suitable technique to perform this proof was described in [28]. This technique consists in proving that the integers \( x' \) and \( x'' \), such that \( x' = x - a \)
Protocol 4 Proof of knowledge of interval checks.

Table 2.2: Intervals used for proofs in \( QR_n \).

<table>
<thead>
<tr>
<th>Value</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>secret value</td>
<td>([-2^{l_\Gamma - k - k'} \ldots 2^{l_\Gamma - k - k'}])</td>
</tr>
<tr>
<td>challenge c</td>
<td>([0 \ldots 2^k] = {0, 1}^k)</td>
</tr>
<tr>
<td>random value r</td>
<td>([-2^{l_\Gamma - 2} \ldots 2^{l_\Gamma - 2}]</td>
</tr>
<tr>
<td>response s</td>
<td>([-2^{l_\Gamma - 1} \ldots - 2^{l_\Gamma - 1}]</td>
</tr>
</tbody>
</table>

and \( x'' = b - x \), are non-negative by representing such values as a sum of four squares. We must take into account that a given value \( x \) has a representation as a sum of four squares if and only if the value is positive. So the prover only has to prove that the values \( x' \) and \( x'' \) can be represented as such sum.

If the sum of four squares of a given non-negative integer \( x \in \mathbb{Z} \) is \( x = x_1^2 + x_2^2 + x_3^2 + x_4^2 \), then we can prove that this representation is correct using the following proof of knowledge:

\[
PK\{(x, x_1, x_2, x_3, x_4) : y = g^x \land y_1 = g^{x_1} \land y_2 = g^{x_2} \land y_3 = g^{x_3} \land y_4 = g^{x_4} \land y = y_1^{x_1} y_2^{x_2} y_3^{x_3} y_4^{x_4}\} \tag{2.1}
\]

2.6 Camenisch-Lysyanskaya (CL) Signature

In order to build credentials we need, besides a commitment scheme, a signature scheme with special properties. The scheme, used in our protocol by the issuer
organizations to grant credentials, is the Camenisch-Lysyanskaya signature [17].
This scheme is suited for our design because it has efficient protocols that allow
to prove the knowledge of a signature.

The signature scheme requires the creation of a public and secret key by the
issuer organization. First, this organization computes a RSA modulus $n$ such
that $n = pq$ and $p$ and $q$ are safe primes, i.e., $p = 2p' + 1$ and $q = 2q' + 1$. The
secret key is the set of the two safe primes. To compute the public key, the
issuer chooses at random $R_0 \ldots R_{L-1}, S, Z \in QR_n$. Thus, we have a public key
$PK = (R_0 \ldots R_{L-1}, S, Z)$ and a secret key $SK = (p, q)$, the parameter $l_n$ is the
length of the modulus $n$.

The signature is used to sign a message, in this case a vector of messages
with the same number of parameters than the number of bases in the public
key. Thus, the message is the set $\{(m_0 \ldots m_{L-1}) : m_i \in \pm\{0, 1\}^{l_m}\}$, where $l_m$
is the maximum length of messages.

To obtain the CL signature the issuer chooses a random prime number $e$
of length $l_e > l_m + 2$ and a random value $v$ of length $l_v = l_n + l_m + l_r$, where $l_r$
is a security parameter. The issuer, making use of its secret key, computes the value $A$
such that $Z \equiv R_0^{m_0} \ldots R_{L-1}^{m_{L-1}} S^v A^e$. Note that to compute $A = \left(\frac{Z}{R_{L-1}^{l_m} S^e}\right)^{\frac{1}{e}}$, we need the modular multiplicative inverse of $e$, i.e., $\frac{1}{e}$. For the computation
of this value the prime factors of $n$ (the secret key) are required. Finally, the
signature of the message $(m_0 \ldots m_{L-1})$ is the tuple $(A, e, v)$.

Entities can verify a CL signature by performing the verification algorithm.
This algorithms takes as inputs the messages and the signature and checks that
$Z \equiv R_0^{m_0} \ldots R_{L-1}^{m_{L-1}} S^v A^e$ and $2^{l_e - 1} < e < 2^{l_e}$ hold.
Chapter 3

The Direct Anonymous Attestation Protocol

In our design of an anonymous credential protocol with smart card support, we have chosen the Direct Anonymous Attestation protocol (DAA) [15] as starting point. This scheme was selected by the Trusted Computing Group [10], an industry standardization body that develops industry standard in the field of trusted computing hardware and software for safety online transactions, to obtain a suitable method for ensuring remote authentication between a hardware module, the trusted platform module (TPM), and a verifier. The main feature of this scheme is that it allows the authentication of the user towards the verifier while preserving the user’s anonymity. In other words, the user can authenticate itself as being in possession of a valid TPM, and so, let the verifier be sure that a message $m$ is sent by such valid TPM without any doubt, but revealing nothing about the user’s identity. The DAA protocol also supports other features, like the possibility of detecting dishonest users that use fake TPM’s, or the linkability of different shows of a credential. In summary, anonymity, detecting rogue users, and possible linkability are provided in this scheme (under the strong RSA and the decisional Diffie-Hellman assumptions).

3.1 Introduction

We chose the DAA protocol as the basis for our credential system with smart card support because it is the first standardized privacy enhancing protocol with anonymous credential features that provides a distinction between a small device with limited resources and a more powerful host computer on the user side.

The components of a TPM are usually implemented in a small devices such as a microcontroller or an Application Specific Integrated Circuit (ASIC) embedded
in a host such as a PC or a mobile device. Due to the limitations of the TPM device, all computations that do not involve the knowledge of the secret values stored on the TPM are outsourced to the host. The host also manages the communication with the verifier. The DAA protocol ensures that a host cannot authenticate without the presence of a TPM. During the protocol these two entities, TPM and host, can be seen as the user or prover that wants to authenticate itself. So in the following if we say "user" we refer to the set of the TPM and the host unless stated different. The verifier, as we mentioned before, is the entity to whom the user wants to authenticate.

The first approach involving these three entities (host, TPM and verifier) to achieve authentication while ensuring the user's privacy, could be the generation of only one secret key stored in all trusted TPMs. In this scenario, when the TPM executes a proof of knowledge with the verifier to prove the knowledge of the secret value, the verifier cannot distinguish if two different requests come from the same TPM. Furthermore, the host does not get any information about the secret value stored in the TPM despite that it can read all messages. The problem of this approach is the vulnerability of the system in case of one TPM being compromised and its secret published. In this case, the verifier is not able to distinguish between trusted and fake TPMs since all of them use the same secret value.

In order to avoid the problem mentioned before, the DAA scheme uses a trusted third party called Privacy Certification Authority (Privacy CA). This party, that we call Issuer Organization, is the only one able to grant the certificate required by the user to authenticate itself towards the verifier. After the user is granted a certificate, she is able to authenticate herself by proving she has the certificate and the secret values on which the certificate is built. Before providing the user with a certificate, the issuer needs to check whether or not the TPM is valid. Thus, it is required that the issuer is able to distinguish between fake and trusted TPMs. The Trusted Computing Group proposes to use a genuine and different Endorsement Key (EK) in each trusted TPM. If we assume that the issuer knows all valid EKs, then the issuer is able to recognize trusted TPMs, see [15] for more details. There exist other methods to distinguish between valid and fake TPMs. In fact, we only need to ensure an authenticated channel between the TPM and the issuer to send only one value. There are several ways to ensure this authenticated channel. However, the use of the Endorsement Keys is the advised method by the TCG.

In summary, we have a trusted TPM with a secret value \( f \) and a valid Endorsement Key known by the issuer. Using the EK to ensure an authenticated channel, the user (in possession of a TPM) and the issuer can perform a two-party protocol to build the certificate (credential) required for further authentication that is finally stored by the user. Besides this certificate, the user must provide a pseudonym \( N_V \) to allow the verifier to detect rogue TPMs (the way of achieving this feature is explained later on). These steps compose the so called Join protocol. Once the user has the certificate, she performs a proof of knowledge to convince the verifier that she has got such a certificate and that she has the secret value used to compute the certificate. This last step, where
the user proves that she has got an anonymous attestation credential and at the same time signs a message $m$ is called DAA-Signing protocol. These protocols (Join and Signing protocol) are explained in detail in the following sections.

It must be taken in account that the DAA protocol serves only as basis of our implementation. As we can see in the next chapter, the scheme we implement is a simplification (in terms of functionality, and so, in the number of parameters required). Moreover, we do not implement a TPM in our system, but we provide smart card support. Note that a TPM is an implementation of a published specification detailing a secure crypto processor, whose complexity is quite bigger than the smart card we use. We also must note that the description of this protocol is based on the first specification published [15]. In this section we explain the DAA scheme as it was designed in [15]. Our system is described in Chapter 5.

Figure 3.1 depicts the architecture of the DAA scheme. The TPM together with the host compose the user.

![Figure 3.1: Architecture of the DAA scheme](image)

### 3.2 High-Level Description

The goal of the DAA is to allow trusted TPMs to get an anonymous attestation credential over a given secret value $f$, and use it to authenticate itself towards the verifier. In this section we provide a description about how the DAA scheme works, and in following sections we explain the two protocols in which this scheme is divided: Join protocol and Signing protocol.
Note that before performing the protocols, the issuer organization must provide a public key $PK_I = (n, g', g, h, S, Z, R_0, R_1, \gamma, \Gamma, \rho)$ and keep hidden a secret key $SK_I = (p, q)$.

In order to obtain a credential, the TPM chooses a secret message $f$ and splits it into two values, $f_0$ and $f_1$, of $l_f$ bits each. This separation simplifies the computations on the side of the issuer. The TPM also chooses an $l_v$-bit value $v'$ and computes the commitment $U := R_0^{f_0} R_1^{f_1} S v' \mod n$. The TPM sends this value to the issuer through the host using an authenticated channel. Note that the host learns nothing about the secret values $f_0$ and $f_1$. The TPM also computes the pseudonym $N_I := \zeta^{f_0 + f_1 2^l} \mod \Gamma$, where the value $\zeta_I$ is derived from the issuer’s name (hence it is known by the issuer).

The pseudonym $N_I$ is not really needed in an anonymous credential system. Its only function is to allow the issuer to recognize the user, and decide if he wants to grant a credential according some security policy. This pseudonym does not open a security hole, because the pseudonym is different for each different tuple user-issuer, in the sense that one user has a different pseudonym for each issuer such that different requests to different issuers can not be linked. Furthermore, the pseudonym $N_V$ used when showing a credential is different, and that avoids the linkability between the issue of a credential and the show of such credential.

By performing a proof of knowledge, the TPM convinces the issuer that the two values $U$ and $N_I$ have been correctly generated, and the issuer decides whether or not he wants to grant a certificate to that pseudonym (the issuer might decide not to grant too many credentials or reissue a credential with respect to a pseudonym previously accepted). Supposing that the TPM fulfills all requirements to get a credential, the issuer picks up a random $l_v$-bit value $v''$, computes $A := \left( \frac{Z^{U S v''}}{U S v'} \right)^{\frac{1}{2}} \mod n$, and sends the tuple $(A, e, v'')$ to the TPM. After performing this steps the host has got the tuple $(A, e)$, called certificate, and the TPM the secret values $(f, v)$.

Finally, the user has got a Camenisch-Lysyanskaya signature [17] on the secret values $f_0$ and $f_1$. The credential is the set of the certificate $(A, e)$ and secret values $(f, v)$ used to compute such credential.

Once the user has a credential she can show it to the verifier to authenticate herself or a message $m$. This is done by performing the so called DAA Signing protocol, which consists in performing a proof of knowledge of the certificate and the secret values it is based on. This is done by computing $A^e R_0^{f_0} R_1^{f_1} S v'' \equiv Z$ and a pseudonym $N_V := \zeta^{f_0 + f_1 2^l} \mod \Gamma$. The first value is used to carry out the proof of knowledge, while the second one is the pseudonym that allows the verifier to detect dishonest users.

When the security of a trusted TPM is broken up and its secret values published, these values are added to a black list. Due to that the verifier knows the value $\zeta$, base of the exponentiation required to compute the pseudonym, the verifier can compute all possible $\hat{N}_V := \zeta^{f_0 + f_1 2^l} \mod \Gamma$ where $(\hat{f}_0, \hat{f}_1)$ are a pair of secret values tagged rogue (and so included in the black list). The verifier uses all entries of the black list to compute the possible rogue pseudonyms and
check whether the pseudonym received matches with one of them. In that case the verifier aborts the protocol.

Besides rogue tagging, the show-credential linkability can be achieved using pseudonyms. We can get this feature by choosing that the value $\zeta$ is computed from the verifier’s name, thus the value $N_V$ will be always the same for a given pair TPM and verifier. This means that all the times that the user shows a credential to the same verifier she uses the same pseudonym, thus different shows are linkable to each other. Otherwise, if we let the user choose the value $\zeta$ randomly, linkability is not provided. We must note that the show-credential does not allow to link the show of different credentials, since the secret values $f_0$ and $f_1$ might change each time the user performs the show protocol. In summary, if we require the TPM to compute the value $\zeta$ from the verifier’s name, we allow the verifier to link different shows of a given credential, but different shows of different credentials belonging the same user are not always linkable (depending on whether or not the TPM changes its secret values). On the other hand if we let the TPM choose the value $\zeta$ at random, then linkability is not provided.

In the DAA protocol all the proofs of knowledge that the user performs to convince the verifier are merged in one proof. This is done by using a proof of knowledge of a representation that proves the knowledge of several secret values using the same challenge for all of them. Using the Fiat-Shamir heuristic [23] we can see this proof as a "signature of knowledge". The concept of signature of knowledge was introduced by Camenisch and Stadler [16]. To understand the concept of a signature of knowledge we first should see a signature $\sigma$ on a message $m$ as the fact of "the owner of a public key PK and its corresponding secret key SK has signed the message $m$". The signature of knowledge is then when "an entity in possession of a witness $w$ to the statement that $x \in L$ has signed a message $m$", thus, that entity is signing the message $m$ under the knowledge of some statement. A more suitable definition is provided by Lysyanskaya in [19]. In practice, the main difference between a proof of knowledge and a signature of knowledge is the non-interactive property of the second one. Therefore, this protocol turns the proofs of knowledge into signatures of knowledge using the Fiat Shamir heuristic. In the reminder, we denote these proofs as $SPK$.

As mentioned before, all operations in the user side that do not involve the secret values are outsourced to the host. The outsourcing of these computations does not open a security hole, because even in the case of computing all operations in the TPM, the TPM has to send the values through the host. Thus, a dishonest host could tag all messages from the TPM to reveal its identity or link its messages. Hence, in the event of having a dishonest host there is nothing we can do, and not outsourcing the operations would not improve the situation. If we outsource the computations that involve the secret values, a dishonest host could publish such secret values, and then, they should be added to the black list (where all the secret values tagged rogue are placed).
3.3 Security parameters

The DAA protocol makes use of the parameters $l_n, l_f, l_e, l'_e, l_v, l_\phi, l_H, l_r, l_\Gamma$ and $l_\rho$ to establish the length of the values used in the protocol. These lengths determine the security provided by the protocol. There exists a trade off between efficiency and security: security improves with bigger lengths but the computations are faster if we reduce the size of the values. The parameter $l_f$ sets the size of the two $f_i$’s in which the secret value $f$ is split (the reason of this split is explained later on). The value $l_n$ is the size of the RSA modulus $n$, $l_e$ is the size of the prime $e$ used in the certificate, while $l'_e$ sets the size of the interval where this value is chosen from. The value $l_v$ establishes the length of $v$, a random value used in the certificate. $l_\phi$ is the parameter that controls the statistical zero-knowledge property. $l_H$ is the output length of the hash function used. $l_r$ is a parameter needed for the reduction in the proof of security. $l_\Gamma$ is the size of the modulus $\Gamma$ and $l_\rho$ the order of the subgroup $\mathbb{Z}_\Gamma^*$ for rogue-tagging. Advisable lengths (in bits) are shown in Table 3.1.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Length (in bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$l_n$</td>
<td>RSA modulus $n$</td>
<td>2048</td>
</tr>
<tr>
<td>$l_f$</td>
<td>secret values $f_1,f_2$</td>
<td>104</td>
</tr>
<tr>
<td>$l_e$</td>
<td>prime number $e$</td>
<td>368</td>
</tr>
<tr>
<td>$l'_e$</td>
<td>interval for $e$</td>
<td>120</td>
</tr>
<tr>
<td>$l_v$</td>
<td>secret value $v$</td>
<td>2536</td>
</tr>
<tr>
<td>$l_\phi$</td>
<td>z-k property</td>
<td>80</td>
</tr>
<tr>
<td>$l_H$</td>
<td>hash function</td>
<td>160</td>
</tr>
<tr>
<td>$l_r$</td>
<td>proof of security</td>
<td>80</td>
</tr>
<tr>
<td>$l_\Gamma$</td>
<td>modulus $\Gamma$</td>
<td>1632</td>
</tr>
<tr>
<td>$l_\rho$</td>
<td>order of $\mathbb{Z}_\Gamma^*$</td>
<td>208</td>
</tr>
</tbody>
</table>

These lengths can be changed while the following equations are fulfilled:

$$l_e > l_\phi + l_H + \max\{l_f + 4, l'_e + 2\}, \quad (3.1)$$
$$l_v > l_n + l_\phi + l_H + \max\{l_f + l_r + 3, l_\phi + 2\}, \quad (3.2)$$
$$l_\rho = 2l_f. \quad (3.3)$$

We can see in Eq. 3.1 that the minimum length for $l_e$ depends directly on the value $l_f$. This is the main reason of splitting the secret $f$ into two smaller values $f_0$ and $f_1$.  


3.4 Issuer’s Public Key

The security of the DAA scheme depends on the correct construction of the issuer’s public key. We have to ensure the correct generation of the key according to the equations described below, otherwise the security requirements do not hold.

The vector of values $PK_I = (n, g', g, h, S, Z, R_0, R_1, \gamma, \Gamma, \rho)$ corresponds to the issuer’s key. In order to build it, the issuer must carry out the following steps:

1. Choose a RSA modulus $n = pq$ where $p = 2p' + 1$ and $q = 2q' + 1$, such that $p, p', q, q'$ are all primes and $n$ has $l_n$ bits.

2. Choose a generator $g'$ of $QR_n$ (the group of quadratic residues modulo $n$) at random.

3. Generate random integers $x_0, x_1, x_z, x_s, x_h, x_g \in [1, p'q']$. This interval is due to that the order of the group $QR_n$ is the product of the two Sophie-Germain primes $p'q'$. With these values compute:

   $$ g := g'^{x_g} \mod n, \quad h := g'^{x_h} \mod n, \quad S := h^{x_s} \mod n, \quad (3.4) $$

   $$ Z := h^{x_z} \mod n, \quad R_0 := S^{x_0} \mod n, \quad R_1 := S^{x_1} \mod n. \quad (3.5) $$

4. Choose two random primes $\rho$ and $\Gamma$ such that $\Gamma = r\rho + 1$, where $\Gamma \in (2^{l_\Gamma - 1}, 2^{l_\Gamma})$ and $\rho \in (2^{l_\rho - 1}, 2^{l_\rho})$. Also, choose a random $\gamma' \in \mathbb{Z}_{\Gamma}^*$ such that $\gamma'^{\frac{\Gamma - 1}{\rho}} \neq 1 \pmod{\Gamma}$. Finally, compute $\gamma := \gamma'^{\frac{\Gamma - 1}{\rho}} \pmod{\Gamma}$.

5. Publish the key $PK_I = (n, g', g, h, S, Z, R_0, R_1, \gamma, \Gamma, \rho)$. The secret key is the tuple of the two safe primes used to compute the modulus $n$, i.e., $SK = (p, q)$.

The user must perform a non-interactive proof in order to verify the key. The values $R_0, R_1, g$ and $h$ must be computed correctly, otherwise the security for the user (anonymity properties) might not hold. It is also important to check the correctness of the value $\gamma$ since if it does not generate a subgroup $\mathbb{Z}_{\Gamma}^*$ the issuer could use this to link different signatures as explained in [15].

3.5 Join Protocol

In the Join protocol the user gets a credential by acquiring a Camenisch-Lysyanskaya (CL) signature $(A, e, v)$ from the issuer on the secret values $f_0$ and $f_1$. The first requirement to perform this protocol is to have an authenticated channel between a trusted TPM and the issuer. Actually, the only condition we have to ensure is that the value $U := R_0^{f_0} R_1^{f_1} S^v \pmod{n}$ (computed by the TPM) is sent by the trusted TPM to the issuer. Otherwise a dishonest host...
could choose two values $\hat{f}_0$ and $\hat{f}_1$, compute a fake value $U$ and send it to the issuer to obtain a credential. As we have mentioned, the issuer can be sure that it is speaking with a valid TPM by having a genuine Endorsement Key in each trusted TPM, and assuming issuer knows all valid EK’s. The way of authenticating a TPM w.r.t. an Endorsement Key is explained in detail in the appendix of [15].

Figure 3.2 illustrates the flow of messages between the user and the issuer in the Join protocol is illustrated. Note that the two signatures of knowledge are not included. The user sends the value $U$ and the pseudonym $N_I$ and receives the tuple $(A, e, v'')$ corresponding to the certificate.

The steps to obtain the credential are the following:

1. The user checks whether or not the $PK_I$ (the issuer’s public key) is valid by performing an authentication with a long-term public key $PK'_I$.

2. The host computes $\zeta_I := (H_{\Gamma}(1||bsn_I))^{(\rho-1)} \mod \Gamma$ and sends $\zeta_I$ to the TPM. The value $bsn_I$ is the issuer’s long term basename, and $H_{\Gamma}$ denotes a hash function $H_{\Gamma}: \{0,1\}^* \rightarrow \{0,1\}^{l_{\Gamma}+l_{\Phi}}$. The TPM checks the correct construction of $\zeta_I$, checking whether the equation $\zeta_I^\rho \equiv 1 \mod \Gamma$ holds.

3. The TPM chooses a secret value $f$ and splits it into two different parts of $l_f$ bits. We use a hash function to compute the value $f$ from the long-term $PK'_I$, a DAA seed previously fixed by the protocol and a counter “cnt”. This counter is increased each time that a given TPM executes the Join protocol, however the TPM can decide to not increase the counter to not change the current pseudonym $N_I$. The value $f$ is computed as follows:

$$f := H(H(\text{DAAseed}||H(PK'_I))||\text{cnt}||0)||\ldots\ldots||H(H(\text{DAAseed}||H(PK'_I))||\text{cnt}||i) \mod \rho,$$

where $i = \lceil \frac{l_{\Gamma}+l_{\Phi}}{l_H} \rceil$. This value $i$ has the value 1 with the security parameters proposed in Sect. 3.3. Therefore, the length of $f$ is $l_H$ (160 bits). We
split this value into $f_0$ (the $l_f$ less significant bits) and $f_1$ (the $l_f$ most significant bits).

The TPM also chooses the value $v' \in \mathbb{R} \{0, 1\}^{l_n+l_e}$ and finally computes $U$ and $N_I$ as follows:

$$U := R_0^{f_0} R_1^{f_1} S^{v'} \pmod{n} \quad N_I = \zeta_0^{f_0+f_1} \pmod{n} . \quad (3.6)$$

The TPM sends these two values to the issuer.

4. As mentioned in Sect. 3.2, the value $N_I$ is used to allow the issuer to follow some security policy for granting credentials. Although we said that we use the pseudonym $N_V$ (used in the Signing protocol) to detect dishonest users, actually this test is performed in both protocols (Join and Signing protocol). In this case the issuer also checks if the received pseudonym $N_I = \zeta_0^{f_0+f_1}$ for all entries $(f_0, f_1)$ in the black list. If the issuer finds a match it aborts the protocol.

5. The TPM proves the knowledge of the secret values $(f_0, f_1, v')$ used to compute the values $U$ and $N_I$ by performing the signature of knowledge as described below:

$$SPK\{(f_0, f_1, v') : U \equiv \pm R_0^{f_0} R_1^{f_1} S^{v'} \pmod{n} \land$$
$$N_I \equiv \zeta_0^{f_0+f_1} \pmod{\Gamma} \land f_0, f_1 \in \{0, 1\}^{l_f+l_e+l_n+2} \land$$
$$v' \in \{0, 1\}^{l_n+l_e+l_n+2} (n_h \parallel n_i) .$$

6. Once the issuer has the values $U$ and $N_I$, it picks up a value $\hat{v} \in \mathbb{R} \{0, 1\}^{l_v-1}$ and a prime $e \in \mathbb{R} [2^{l_v-1}, 2^{l_v-1} + 2^{l_v}]$. Then he computes $v'' := \hat{v} + 2^{l_v-1}$ and the value:

$$A := \left( \frac{Z}{US^{v''}} \right)^{\frac{1}{e}} \pmod{n} . \quad (3.7)$$

Note that in order for the $e$th-root to be feasible, it is completely required the knowledge of the two safe primes $p$ and $q$ used to compute the modulo $n$ (using the Extended Euclidean Algorithm [29]). Since both $p$ and $q$ are kept secret by the issuer, only the issuer can compute the value $A$. Therefore, the issuer is the only one able to compute the CL-Signature.

7. The issuer sends the values $A$, $e$ and $v''$ to the user and performs the following proof to prove the value $A$ was correctly computed:

$$SPK\{(d) : A \equiv \pm \left( \frac{Z}{US^{v''}} \right)^{d} \pmod{n} \}(n_h)$$

The value $n_h$ attached in the proof is a nonce to perform the challenge required to carry out such a proof. The way of performing this proof and
the previous one can be found in more detail in [15]. The main reason to perform this proof of knowledge is to ensure that the value $A$ is correctly computed, otherwise this value could be linkable to the value $T_1 = Ah^w$ (computed in the Signing protocol).

8. The user receives the values $(A, e, v'')$. The values $A$ and $e$ are stored in the host, while the value $v''$ is forwarded to the TPM. Then, the TPM computes $v = v' + v''$ and stores the secret values $(f_0, f_1, v)$. These secret values are required in the Signing protocol to prove that the user has the values on which the credential is built.

### 3.6 Signing protocol

Using the DAA Signing protocol, performed between the user (host and TPM) and the verifier, the user can authenticate herself and at the same time sign a message $m$. The goal is to authenticate the message $m$ by proving the user has an anonymous attestation credential. The protocol must be performed after obtaining the inputs $(n, R_0, R_1, S, \Gamma, \rho)$ and $(f_0, f_1, v)$ in the TPM, as well as the certificate $(A, e)$ and the public key $(n, g, g', h', R_0, R_1, S, Z, \gamma, \Gamma, \rho)$ in the host.

In Figure 3.3 the messages sent between the user and the verifier in the Signing protocol are illustrated.

![Figure 3.3: Simplified model of the Signing protocol](image)

The Signing protocol is explained in detail below:

1. The host computes the base for the pseudonym, depending on the request of the verifier, and sends it to the TPM. There are two different ways of obtaining the value $\zeta$: at random or deriving it from the verifier’s name. The latter allows the show-credential linkability.

   \[ \zeta \in_R \langle \gamma \rangle \quad \text{or} \quad \zeta := (\mathcal{H}_\Gamma(1||bsn_v))^{(\Gamma-1)} \rho \mod \Gamma. \quad (3.8) \]

The TPM checks whether $\zeta^\rho \equiv 1 \mod \Gamma$ and computes the pseudonym $N_V := \zeta^{f_0+f_1} \mod \Gamma$. 
3.6. SIGNING PROTOCOL

2. The host generates random integers \( w, r \in \{0, 1\}^{l_a+l_e} \) and computes \( T_1 := A h^w \mod n \) and \( T_2 := g^w h^r (g')^r \mod n \). This way the values \( A \) and \( e \) are kept hidden, the verifier can not extract them from the \( T_1 \)'s values.

3. The main step of this protocol is the "signature of knowledge" performed between the user and the verifier. In this step the user proves to the verifier it has an anonymous credential.

\[
SPK((f_0, f_1, v, e, w, r, e w, e e, e r)) : \\
Z \equiv \pm T_1^{f_0} R_0^{f_1} R_1^{f_1} S^w h^{-e w} \mod n \land T_2 \equiv \pm g^w h^r (g')^r \mod n \\
1 \equiv \pm T_2^{-e} g^w h^r (g')^r \mod n \land N_v := \zeta^{f_0+f_1+e r} \mod \Gamma \\
f_0, f_1 \in \{0, 1\}^{l_f+l_o+l_{h+2}} \land (e - 2^e) \in \{0, 1\}^{l_e+l_o+l_{h+1}} (n_v||n_e||b||m) \tag{3.9}
\]

The values \((n_v||n_e||b||m)\) are attached to build the challenge, \( n_v \) and \( n_e \) are nonces, \( m \) is the message we want to sign, and \( b \) describes the use of the protocol (0 if \( m \) is generated by the TPM and 1 if the message is an input to the TPM). Note that the computations at the user side are distributed between the host and the TPM in such a way that only the computations that involve the secret values \( f_0, f_1 \) and \( v \) are executed in the TPM. All the computations required to carry out this proof are described below.

(a) The TPM chooses three integers at random:

\[
r_v \in R \{0, 1\}^{l_v+l_o+l_i} , \quad r_{f_0}, r_{f_1} \in R \{0, 1\}^{l_f+l_o+l_i} ,
\]

and computes:

\[
\hat{T}_1 := R_0^{f_0} R_1^{f_1} S r_v \mod n \tag{3.11} \\
\hat{r}_f := r_{f_0} + r_{f_1} 2^{l_f} \mod n \tag{3.12} \\
\hat{N}_v := \zeta^{e r} \mod \Gamma . \tag{3.13}
\]

Both values \( \hat{T}_1 \), and \( \hat{N}_v \) are sent to the host.

(b) The host picks up the following random integers:

\[
r_e \in R \{0, 1\}^{l_e+l_o+l_h} , \quad r_{e e} \in R \{0, 1\}^{l_e+l_o+l_h+1} , \\
r_r, r_v \in R \{0, 1\}^{l_r+l_o+l_i} , \quad r_{e e}, r_{e r} \in R \{0, 1\}^{l_e+l_o+2l_a+l_h+1} ,
\]

and computes the values:

\[
\hat{T}_1 := \hat{T}_1^{r_v} T_1^{r_v} h^{r_v} \mod n , \tag{3.15} \\
\hat{T}_2 := g^{r_v} h^{r_v} (g')^r \mod n , \tag{3.16} \\
\hat{T}_2 := \hat{T}_2^{r_v} g^{r_v} h^{r_v} (g')^r \mod n . \tag{3.17}
\]
(c) The host computes the hash value $c_h$ and sends it to the TPM:

$$c_h := \mathcal{H}(n||g||g'||h||R||S||Z||\gamma||\Gamma||\rho) || \zeta || (T_1||T_2) || (\tilde{T}_1||\tilde{T}_2) || N_V || n_v .$$  

(3.18)

(d) The TPM chooses a nonce $n_t \in \{0,1\}^l$ randomly and computes the challenge $c$ with the received value $c_h$.

$$c := \mathcal{H}(c_h||n_t)||m) \in [0,2^{lH-1}] .$$  

(3.19)

It sends these two values $(n_t,c)$ to the host.

(e) The s-values required to perform the validation of the proof are built as follows. Note that $s_v$, $s_{f_0}$ and $s_{f_1}$ are computed in the TPM since it is completely required to have the knowledge of the secret values, however the rest of s-values are computed in the host.

The host computes:

$$s_v := r_v + cw , \quad s_{f_0} := r_{f_0} + cf , \quad s_{f_0} := r_{f_0} + cf .$$  

(3.20)

The host computes

$$s_e := r_e + c \cdot (e - 2^{l_r-1}) , \quad s_{ee} := r_{ee} + ce^2 , \quad s_w := r_w + cw ,$$

$$s_{ew} := r_{ew} + cwe , \quad s_r := r_r + cr , \quad s_{er} := r_{er} + cer .$$  

(3.21)

The host is now able to build the signature $\sigma$ as follows:

$$\sigma := (\zeta, (T_1, T_2), N_V, c, n_t, (s_v, s_{f_0}, s_e, s_{ee}, s_w, s_{ew}, s_r, s_{er})) .$$  

(3.22)

(f) Once the signature is built and sent to the verifier, the "verification algorithm" must be performed to validate it. The inputs to the verifier are the signature $\sigma$, the public key of the issuer, and the message $m$ to authenticate. The verifier carries out the operations described below:

$$\tilde{T}_1 := Z^{-c} T_1^{s_v+c2^{l_e-1}} R^{s_f} S^{s_{f_0}} h^{s_{ew}} \mod n ,$$  

(3.23)

$$\tilde{T}_2 := T_2^{-c} g^{s_w} h^{s_{ew}} \mod n ,$$  

(3.24)

$$\tilde{T}_2' := T_2^{-(s_e+c2^{l_r-1})} g^{s_{ew}} h^{s_{ee}} g^{s_{er}} \mod n ,$$  

(3.25)

$$\tilde{N}_V := N_V^{-c} \zeta^{s_{f_0}} + s_{f_1} 2^l \mod n .$$  

(3.26)

The verifier checks whether or not the following equations hold.

$$c = \mathcal{H}(\mathcal{H}((n||g||g'||h||R_0||R_1||S||Z||\gamma||\Gamma||\rho) || \zeta || (T_1||T_2))||N_V||$$

$$||(\tilde{T}_1, \tilde{T}_2, \tilde{T}_2')||\tilde{N}_V || n_v || n_t || b || m) .$$  

(3.27)

$$s_{f_0}, s_{f_1} \in \{0,1\}^{l_f+l_\sigma+l_H+1} , \quad s_e \in \{0,1\}^{l_e+l_\sigma+l_H+1} \quad (3.28)$$
In case that the show-credential linkability feature is enabled (the value $\zeta$ is not chosen at random) the verifier has to check whether the base $\zeta$ of the pseudonym $N_V$ is built in the correct way. That means the received value $\zeta$ has to match with $(\mathcal{H}_1(1||bsn_v))^\frac{(\Gamma-1)}{\rho} \mod \Gamma$ (see eq. 3.8). Anyway, the verifier has to check the black list to find a possible match of the pseudonym with some pair of secret values $(\hat{f}_0, \hat{f}_1)$ tagged rogue. Rogue tagging can be performed faster if the base $\zeta$ was not chosen at random, since all possible rogue pseudonyms $\hat{N}_V := \zeta^{\hat{f}_0 + \hat{f}_1 2^i}$ can be precomputed. Once the verification algorithm is finished and succeeded the verifier knows that the message $m$ comes from the correct TPM.
Chapter 4

Smart Cards and the Java Card Standard

In this chapter first we give some general information about smart cards, and secondly we explain some necessary preliminaries about the Java Card standard. The information in this chapter is based on versions 2.1.1 [11] and 2.2.1 [12] of the standard.

4.1 Smart Card Basics

A smart card is typically a small device, having the size and shape of a credit card. Its most basic representation, as shown in figure 4.1, is as an integrated circuit embedded in a plastic body. Physical characteristics of these cards are described in the ISO 7816-1 specification [5]. Smart cards are portable, tamper resistant devices providing support for enhanced security systems in a variety of application domains. They provide possibilities for data storage and processing. Due to their low price, they have become a popular kind of cryptographic devices over the past few decades.

Smart cards are used in many different domains where strong security mechanisms are desired in a flexible way. The application domain the majority of people are most familiar with is banking. Banking cards (credit as well as debit) are the most obvious example of smart cards. These banking cards often not only provide debit and credit functions, but can also include some kind of e-wallet system for electronic cash payments. Also, the banking cards can be used to provide access to an online account management system. Another major application of smart cards can be found in the telecommunication sector. First, smart cards were mainly used to provide an easy and secure payment system for public phone usage. Now one of their main applications is as Subscriber Identity Modules (SIM) in mobile GSM phones. Also many access control systems are
designed with smart card support. Examples can be found in physical access control systems (to buildings, public transport, etc.), as well as secure websites or parts of internal networks. A rather new application of smart cards is the concept of Electronic ID (EID) cards, as a replacement for paper identification documents.

A major advantage of smart cards over classical magnetic stripe cards are the built-in processor and memory. The processing and storage capabilities of the cards allow to add important security features. For example, certain data (like secret keys) can be set up to never leave the card, but instead be used to execute cryptographic protocols for encrypting exchanged data, signing data, etc. When used in the correct way, smart cards are supposed to be a lot harder to break into than a regular personal computer.

4.1.1 Hardware

For the description of the hardware involved in smart card systems, we first have to make an important distinction between contact and contactless cards. The difference between both groups is their interface to the outside. The classical (contact) smart cards have a metal contact interface placed on the card as shown in figure 4.1, whereas contactless cards use an antenna and radio waves to communicate with a special reader. Advantages of contactless cards are that they allow faster and easier transactions, since they do not have to be placed in a reader in the correct way, but rather just brought inside a region around it. Since the cards used throughout this work are contact cards, details about contactless smart card technology are omitted. The interested reader is referred to [35] or [20].

Contact smart cards communicate with a host through a Card Acceptance Device (CAD). A CAD can e.g. be connected to a computer with a USB interface. Figure 4.2 shows the contacts a CAD uses to connect to a smart card. The different contacts are:

- **Vcc**: the power supply for the card, this is usually 3 or 5 volts
4.1. SMART CARD BASICS

![Contact Points Table]

- **GND**: the reference – ground – voltage
- **RST**: used to send a reset signal to the card, resulting in a warm reset
- **CLK**: the external clock signal
- **I/O**: the data and command transfer contact point for half-duplex communication
- **Vpp**: programming power supply for first generation cards
- The two remaining contact points are reserved for future use (RFU)

For a detailed specification of the dimensions and locations of these contacts, see the ISO 7816-2 specification [6].

Commands can be sent from the host to the card, resulting in the communication scheme of figure 4.3. Section 4.1.2 provides more details about smart card communication protocols.

The Central Processing Unit (CPU) found in a typical smart card is an 8, 16 or 32 bit controller, running on an external clock of 4 to 6 MHz, but possibly with an internal clock multiplier of e.g. 2, 4 or 8. Many modern smart cards also comprise a coprocessor to execute cryptographic operations efficiently.

Three different kinds of memory are usually present:

- **Read-only memory (ROM)**: No power is necessary to hold the data in this type of memory. Since it can be written only once, it contains the fixed part of a smart card’s program code.

- **Electrically erasable programmable read-only memory (EEPROM)**: Like for ROM, no power is needed to keep memory contents. The difference is that changing memory contents is possible. Therefore EEPROM is used for storage of user application code and data. A limitation exists though on the number of times it can be rewritten.

- **Random access memory (RAM)**: As opposed to (EEP)ROM, this is the volatile part of a smart card’s memory. It is the fastest kind of memory available in smart cards, and can be accessed an unlimited number of times. It is used to store temporary data.
4.1.2 Communication Model

A smart card never functions as a stand-alone system, but is always part of some distributed system. The smart card can be seen as the client in these systems, with another device – like a computer, a banking terminal, etc. – acting as the host. The communication model used is a master-slave model, where the smart card (slave) stays passive as long as no command is received from the host (master). Obviously a standardised means of communication between both devices is necessary. As already explained, the high-level communication mechanism can be visualised as in figure 4.3: the host sends commands to the card, and it always expects a response. Communication happens in half-duplex mode, meaning data can be sent in both directions, but not at the same time.

The data packets used for smart card communication are called Application Protocol Data Units (APDUs), where each APDU represents either a command or a response message. The APDU protocol is specified in the ISO 7816-4 standard [7], and describes the structure of both command (C-APDU) and response (R-APDU) messages. Figures 4.4 and 4.5 illustrate the structure of both APDU types. Table 4.1 explains the different fields present in APDUs.

<table>
<thead>
<tr>
<th><strong>Header</strong> (mandatory)</th>
<th><strong>Body</strong> (optional)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLA</td>
<td>INS</td>
</tr>
</tbody>
</table>

Figure 4.4: The structure of a command APDU.

<table>
<thead>
<tr>
<th><strong>Body</strong> (optional)</th>
<th><strong>Trailer</strong> (mandatory)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data</td>
<td>SW1</td>
</tr>
</tbody>
</table>

Figure 4.5: The structure of a response APDU.

4.2 Java Card

The purpose of the Java Card standards [12, 11] is to allow programs written in the Java programming language to run on resource constrained devices like
4.2. JAVA CARD

Table 4.1: The different fields in command and response APDUs.

<table>
<thead>
<tr>
<th>Field</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data</td>
<td>The data field in a command or response APDU.</td>
</tr>
<tr>
<td>CLA</td>
<td>Class byte.</td>
</tr>
<tr>
<td>INS</td>
<td>Instruction byte.</td>
</tr>
<tr>
<td>P1, P2</td>
<td>Parameter bytes.</td>
</tr>
<tr>
<td>Lc</td>
<td>The length of the data field in a command APDU (length command).</td>
</tr>
<tr>
<td>Le</td>
<td>The length of the data the host expects as response (length expected).</td>
</tr>
<tr>
<td>SW1, SW2</td>
<td>Status words of a response APDU.</td>
</tr>
</tbody>
</table>

smart cards. In this way, advantages of regular Java, like object oriented programming (OOP) and code portability can be applied to smart cards. Java Card also allows a (single-threaded) multi-application system to run on smart cards. The Java Card standard specifies a secure mechanism to load third-party applications (or applets in the context of Java) post-issuance, while preserving security by means of standard Java mechanisms (e.g. no pointers, checking of array bounds, etc.), strengthened by a firewall mechanism.

A single application running on a Java Card platform is called an applet, but it is only possible to upload packages, where each package possibly contains multiple applets. Each applet (and also each package) has a unique identifier, called the Application Identifier (AID). Before commands can be sent to a specific applet, the applet has to be selected with a select APDU containing the AID.

Of course running Java with its full capabilities on a smart card is not possible, so Java Card technology represents a trade-off between offering the full flexibility of Java on one hand, and limiting resource usage on the other hand. The solution is to let Java Card platforms offer only a subset of the Java language features (see section 4.2.1), and to split the Java virtual machine (JVM) into two parts (see section 4.2.2).

4.2.1 Java Language Subset

To be able to run on resource-constrained devices, Java Card does not offer all the functionality offered by the standard Java language. Table 4.2 gives an overview of the functionality differences between Java and Java Card [20].

4.2.2 Java Card Converter

To limit the amount of resources needed for the Java Card Virtual Machine (JCVM), it is implemented as two separate parts. The first part, the Java Card converter, runs off-card on a PC before uploading the code to the card. The
Table 4.2: Supported and unsupported Java language constructs in the Java Card standard.

<table>
<thead>
<tr>
<th>Supported</th>
<th>Unsupported</th>
</tr>
</thead>
<tbody>
<tr>
<td>small primitive data types (boolean, byte, short, and optionally int)</td>
<td>large primitive data types, characters and strings</td>
</tr>
<tr>
<td>one-dimensional arrays</td>
<td>multi-dimensional arrays</td>
</tr>
<tr>
<td>Object Oriented (OO) Programming features:</td>
<td>Dynamic class loading</td>
</tr>
<tr>
<td>packages, classes, interfaces, inheritance, virtual methods, overloading, access scopes, exceptions, etc.</td>
<td>Garbage collection and finalization</td>
</tr>
<tr>
<td>Dynamic class loading</td>
<td>Multithreading</td>
</tr>
<tr>
<td>Garbage collection and finalization</td>
<td>Security manager</td>
</tr>
<tr>
<td>Multithreading</td>
<td>Object serialization</td>
</tr>
<tr>
<td>Security manager</td>
<td>Object cloning</td>
</tr>
</tbody>
</table>

The goal of the Java Card converter is to take some of the time-consuming tasks of the JVM away from the card, like checking the well-formedness of the classes. It also slightly optimises the bytecode, and resolves symbolic references. The converter processes all the class files in one package at once, producing a .CAP file and a .EXP file for each package. The CAP file contains the code to actually be uploaded to the card, and the EXPort file is used to resolve references to other packages.

The Java Card bytecode interpreter running on the card is then only left with the essential tasks to be executed on-card: executing the bytecode instructions, controlling memory allocation, and ensuring runtime security.
4.2. JAVACARD

4.2.3 Java Card Memory

As explained, a typical smart card has three kinds of memory: ROM, EEPROM, and RAM. The JCRE code (i.e. the virtual machine and API packages) is usually stored in ROM. The Java Card runtime stack and so-called transient objects are allocated in RAM, while EEPROM is used for applet code and persistent objects. Because memory (volatile as well as non-volatile) on smart cards is limited, and garbage collection is not included in the Java Card standards, objects should be created only once during an applet’s lifetime. Therefore references to transient objects should be stored in persistent memory.

- **Persistent objects** are allocated in EEPROM and thus preserve data when power is removed from the card. Updates to single fields in a persistent object are atomic, i.e. no undefined state is created when power is lost during the update.

- **Transient objects** are allocated in RAM, but a reference to them should be kept in EEPROM, because otherwise the memory space becomes unreachable and cannot be reused.

4.2.4 Java Card Crypto API

Table 4.3: Class summary of the cryptography-related classes in the Java Card API [11]

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>package javacard.security</td>
<td></td>
</tr>
<tr>
<td>KeyBuilder</td>
<td>a key object factory</td>
</tr>
<tr>
<td>KeyPair</td>
<td>a container for a key pair (private and public key)</td>
</tr>
<tr>
<td>MessageDigest</td>
<td>the base class for hashing algorithms</td>
</tr>
<tr>
<td>RandomData</td>
<td>the base class for random number generation</td>
</tr>
<tr>
<td>Signature</td>
<td>the base class for signature algorithms</td>
</tr>
<tr>
<td>package javacardx.crypto</td>
<td></td>
</tr>
<tr>
<td>Cipher</td>
<td>the base class for cipher algorithms</td>
</tr>
</tbody>
</table>

A smart card is almost always at least partly used to execute some cryptographic protocols. Java, however, is a very inefficient programming language to write complex algorithms that perform calculations on large numbers. Therefore, the Java Card standard includes an Application Programming Interface (API) offering cryptographic functionality. In this way, manufacturers of Java Card platforms can implement cryptographic functionality on a dedicated coprocessor in a very efficient way or use optimised libraries, offering only the high-level functions through the crypto API.
Table 4.3 gives an overview of the available classes and what they are used for. Classes are divided into two packages. Most classes for the Java Card security framework are supplied by the package javacard.security, classes and interfaces for export-controlled functionality are contained in the extension package javacardx.crypto [11].

The following list gives an overview of the cryptographic functionality included in the Java Card standards 2.1.1 and 2.2.1:

- **Message digests**: MD5, RIPEMD160, SHA-1
- **Random data generation**: pseudo random and secure random
- **Ciphers**: triple DES, DSA, RSA
Part II

Design
Chapter 5

Designing an Anonymous Credential System with Smart Card Support

The goal of this chapter is the design of an anonymous credential system where users can obtain credentials from trusted organizations and prove the possession of such credentials afterwards. These systems have been proposed as the best known way to provide privacy properties. This kind of systems are rising in demand, and different implementations are emerging to achieve the consistency of credentials, a term used to refer the basic security properties in credential systems. Consistency refers to the fact that it is not possible “to forge a credential for a user even if users and other organizations team up and launch an adaptive attack on the organization” [18].

In the following sections we explain how, departing from the DAA scheme, we achieve a simpler credential system that allows users to authenticate towards a verifier, while holding the anonymity property and eliminating the possibility of forging credentials. We explain the simplifications made to achieve this extendable implementation, and the security issues raised by the simplifications made.

The first decision of our system is the use of a smart card instead of a complex TPM as the secure cryptographic module. Note that, although both smart cards and TPMs can provide security services and both can be supplied with cryptoproducts, the TPM follows a more complex published specification. In the following, when we say “card”, we refer to a smart card.
5.1 Simplifications

The simplifications presented in this section help the construction of a more extendable program, and at the same time allow an easier implementation free of errors. A wild protocol specification without superfluous features is shown in Sect. 5.2. Note that all simplifications we make imply consequences in the security model of our system. Some of those consequences must be taken into account in further improvements in order to achieve a complete and secure system.

5.1.1 Pseudonyms

The main simplification we make in the original DAA scheme is removing all computations and parameters that deal with pseudonyms, i.e., both $N_I$ (pseudonym towards the issuer) and $N_V$ (pseudonym towards the verifier), are removed. As a consequence, the issuer’s key (see Sect. 3.4) and the signature $\sigma$ (see Eq. 3.22) are significantly reduced.

This simplification implies losing the ability of detecting rogue cards present in the original DAA protocol. Besides, a policy to grant credentials cannot be designed, since users are not distinguishable under a pseudonym. However, an anonymous credential system does not require to have these features under the point of view of the consistency of credentials, as pseudonyms do not prevent the possibility of forging credentials. Furthermore, there exist other suitable solutions to implement pseudonyms besides the one described in the DAA scheme. The inclusion of pseudonyms in our system is left as a line of future work.

By removing the pseudonyms $N_I$ and $N_V$, both the Join protocol (see Sect. 3.5) and the Signing protocol (see Sect. 3.6) are significantly reduced. Examples of what this simplification implies are the reduced public key $PK_I = (n, g, g', h, R_0, R_1, S, Z)$ and the signature

$$\sigma := ((T_1, T_2), c, n, (s_v, s_{f_0}, s_{f_1}, s_r, s_{ee}, s_w, s_{ew}, s_{er})), \quad (5.1)$$

with respect to the ones described in the original DAA protocol.

5.1.2 The secret value $f$

In order to facilitate the operations in the card some simplifications need to be made, even if they do not make sense in the host side. This is the reason for not splitting the secret value $f$ into smaller values $f_0$ and $f_1$ (see Sect. 3.5). As described in the original protocol (see Eq. 3.6) the value $f$ is computed by using a hash function on some variables. This hash function outputs a value $f$ of $l_H$ bits that is split in two $l_f$-bit values $f_0$ and $f_1$ in the original scheme. The reason for splitting the value $f$, is to reduce the size of the prime $e$ of the certificate (see Eq. 3.1). However, each secret value $f_i$ implies the generation of an integer $r_{f_i}$ used to compute a value $T_1$, (see Eq. 3.11). According to this equation, the card needs to perform three exponentiations. By not splitting the value...
5.1. SIMPLIFICATIONS

The number of exponentiations in the card is reduced to only two: $\tilde{T}_1 := R^f S^{\tau} \pmod{n}$. This simplification increases the length of the prime $e$ that belongs to the certificate, so there exists a trade off between the time required to compute the prime $e$ in the computer and the time that the card takes to compute the multiexponentiation. Nevertheless, due to the severe limited resources in the card we must take the decision to favour the card.

In our design, rather than computing the value $f$ with a hash function, we only generate the number at random, ensuring that its length matches with the output length of the hash function used in the protocol. This way the values used in the hash function (the counter, the seed and the long-term $PKI'$) are not required.

This simplification also affects the composition of the issuer's public key. The values $R_0$ and $R_1$ are used as basis of the exponentiations to compute the values $R_0^f$ and $R_1^f$. By having only one $f$ the key only needs one $R$. Therefore, the issuer's public key is reduced to $PK_I = (n, g, g', h, R, S, Z)$.

5.1.3 Security parameters

We can say that a system is secure if and only if the best known algorithm to break the system requires a not-polynomial time depending on some security parameter. This means that, with a given algorithm to attack the system and a value of the security parameter of such a system, if we increase the value of the parameter there is not an upper bound for the time that the algorithm needs to break the security. Of course, with a given value for such parameter we have an upper bound for the time required for the attacker. Therefore, the security increases by using bigger security parameters, but with smaller parameters the computations are faster, because these security parameters set the size of the values used in the protocol. There exists a trade-off between security and the speed of computations.

In order to speed up our design we decide to reduce the security parameter $l_n$ with respect the value described in Sect. 3.3. Therefore, the values of all parameters that directly depend on $l_n$ are changed as well. All computations carried out during the protocol are operations whose time directly depends on the lengths of such parameters. Mostly, smaller parameters result in a reduction of the time required for the card. Thus, the best way to make all computations faster is by reducing this parameters. The only drawback of this approach is that the security of the scheme goes down.

Table 5.1 shows both the suggested lengths in our protocol and the original DAA lengths.

As we can see, the value $l_n$ is halved. The value $l_v$, which depends on the value $l_n$, is also significantly reduced. However, the value $l_e$ is slightly increased due to its dependency on the value $l_f$ that now has the same vale as the output of the hash function used in the protocol, $l_H$. The three parameters related to the computation and use of pseudonyms are removed, while the rest conserve their previous values.
Table 5.1: Suggested lengths in the protocol simplification

<table>
<thead>
<tr>
<th>Parameter</th>
<th>New values</th>
<th>Original values</th>
</tr>
</thead>
<tbody>
<tr>
<td>$l_n$</td>
<td>1024</td>
<td>2048</td>
</tr>
<tr>
<td>$l_f$</td>
<td>160</td>
<td>104</td>
</tr>
<tr>
<td>$l_e$</td>
<td>410</td>
<td>368</td>
</tr>
<tr>
<td>$l'_e$</td>
<td>120</td>
<td>120</td>
</tr>
<tr>
<td>$l_v$</td>
<td>1512</td>
<td>2536</td>
</tr>
<tr>
<td>$l_\phi$</td>
<td>80</td>
<td>80</td>
</tr>
<tr>
<td>$l_H$</td>
<td>160</td>
<td>160</td>
</tr>
</tbody>
</table>

5.1.4 Changes in the signature of knowledge

A closer look to the equations of the DAA Signing protocol (see Sect. 3.6) shows that some variables included in the signature of knowledge are not really needed to fulfill the security properties. An example of this, corresponds to the parameter $T_2$ (see Sect. 3.6, Eq. 3.16). The couple of values $T_1$ and $T_2$ are commitments on the values of the certificate $(A,e)$. These values are used by the host to hide the values of the certificate $(A,e)$ while performing the signature of knowledge in the Signing protocol. The value $T_1 = Ah^w (\mod n)$ hides the value $A$ using the value $h$ that belongs to the issuer’s key and the integer $w$, computed at random in the host. The value $T_2 = g^w h^e g^{r'} (\mod n)$ hides that value $w$ and the value $e$ of the certificate. However, the computation of the value $T_2$ without the second exponentiation $h^e$ (i.e., $T_2 = g^w g^{r'} (\mod n)$) allows a simplification in the signature of knowledge and the security properties hold.

We can see in the signature of knowledge that, despite removing the value $h^e$, the knowledge of the value $e$ is still proved.

After this simplification the new and reduced ”signature of knowledge” used in the Signing protocol is:

$$SPK\{ (f,v,e,w,r,ew,er) :$$

$$Z \equiv \pm T_1^e R^f S^v h^{-ew} (\mod n) \land T_2 \equiv \pm g^w (g')^r (\mod n) \land$$

$$1 \equiv \pm T_2^{-e} g^{ew} (g')^{er} (\mod n) \land f \in \{0,1\}^{l_f + l_\phi + l_H + 2} \land$$

$$(e - 2^{l_e}) \in \{0,1\}^{l'_e + l_\phi + l_H + 1} (n_t || n_v || m)$$

5.1.5 Refactoring

As a result of all simplifications described until now, we obtain a system that fulfills all requirements, i.e., the ability of building and showing credentials as well as the anonymity property. The Signing protocol (called as well Show
protocol) is implemented in the same way as in the DAA scheme (with the simplifications mentioned).

The next step to achieve a better approach to a real credential system must be the implementation of a real issuer organization. Otherwise we would have to generate the public and secret key and compute the credential in the host. Thus, in the final design of our system we cover the implementation of the issuer.

Moreover, the new specification about the DAA scheme (not published yet), source of our work, provides simplifications in the protocol that substantially modify the design with respect to the previous specification. Therefore, the final design of our protocol, presented in the following section, describes a more straightforward way to perform the "signature of knowledge" in the Signing protocol. This change allows a reduction in the number of computations needed in such protocol.

5.2 Final Design

The math specification of our system results from the simplifications about the DAA scheme we described in previous sections. Thus, the security of our system is based on the same assumptions as the original DAA scheme. Although some of our decisions to simplify the protocol reduce such security, we only made simplifications taking into account their possible consequences.

In this specification we take into account that our design covers the implementation of not only the host and the card, but also a real verifier and issuer organization. We explain the generation of the public and secret key in the issuer organization, the Join protocol to grant credentials and the Signing protocol (also called Show protocol) to show credentials.

5.2.1 Key Generation

In the new issuer’s public key, there are three missing values with respect to the previous specification. The new key has four values \( (n, S, R, Z) \), and the bases \( g, g', h \) do not exist anymore. Like in the previous specification we removed all values that deal with the management of pseudonyms. In order to generate the key, the issuer proceeds as follows:

1. Computes an integer \( n \in \{0,1\}^{l_n} \) such that \( n = pq \) where \( p \) and \( q \) are safe primes, i.e., \( p = 2p' + 1 \) and \( q = 2q' + 1 \), where \( p,p',q,q' \) are prime numbers. The value \( n \) is the special RSA modulus of the protocol and \( QR_n \) its quadratic residues group.

In order to generate the modulus \( n \) of a given length \( l_n \) (in bits) we use a function that directly outputs a Sofie Germain prime of a given length such that we have not to make an exhaustive search for finding two primes \( p \) and \( q \) that fulfills \( p = 2p' + 1 \) and \( q = 2q' + 1 \) and \( n = pq \) has \( l_n \) bits. The way to generate \( n \) is to get the two Sofie Germain primes \( p' \) and \( q' \) of \( (l_n/2) \) bits, using the function mentioned, and computing the modulus
CHAPTER 5. DESIGNING AN ANONYMOUS CREDENTIAL SYSTEM

\[ n \text{ as } n := (2p' + 1)(2q' + 1). \] Thus, \( n \) has a length between \( l_n \) and \( l_n + 1 \), depending on the values outputted by the function. In case of having a modulus of \( l_n + 1 \) bits we would obtain an slight improvement in the security of the scheme.

2. Chooses randomly a generator \( S \) of the group \( QR_n \).

Although there exist suitable algorithms for a fast computation of a generator of \( QR_n \), we just choose at random one value that belongs to \( \mathbb{Z}_n \) (in the range \([1, n]\)) and compute its square and the modular reduction. In other words, we choose a value \( x \in [1, n] \), and obtain \( S := x^2 \pmod{n} \). This is a valid approach since the probability that a value, chosen at random in \( QR_n \), is not a generator is negligible. Thus, we do not need any test or algorithm to check whether or not the value \( S \) computed is a generator.

3. Chooses randomly two integers \( x_0, x_z \in [1, p'q'] \) and computes the following values:

\[ Z := S^{x_z} \pmod{n}, \quad \text{(5.2)} \]
\[ R := S^{x_0} \pmod{n}. \quad \text{(5.3)} \]

The integers \( x_0 \) and \( x_z \) are the exponents of an exponentiation with base \( S \). As \( S \) is a generator of \( QR_n \), the values \( Z \) and \( R \) also belong to \( QR_n \).

The values \((n, S, R, Z)\) form the public key of the issuer organization, while the safe primes are the secret key \( SK = (p, q) \). Note that the knowledge of \( SK \) allows the construction of the certificate \((A, e)\), which is built in the Join protocol explained in the next section.

According to the original math specification the host obtains the public key from the issuer and the verifier receives the public key from the host when the host sends the signature. Nevertheless, we decide to send this key (from the host to the verifier) at the beginning of the protocol, when the host sets up the communication with the verifier.

5.2.2 Join Protocol

By performing the Join protocol, the user (host and card) obtains a credential from the issuer organization. In summary, the host together with the card obtains a Camenisch-Lysyanskaya signature computed by the issuer using its secret key. We implement this protocol following the same specification as the original DAA scheme. However, the two proofs of knowledge required in this protocol are not carried out in order to keep it as simple as possible. Thus, the only simplifications made in this protocol with respect to the original one, consist on removing both the proofs of knowledge and all parameters that deal with pseudonyms.

The messages forwarded while performing the protocol are illustrated in Figure 5.1. The keys \( PK \) and \( SK \) are known before starting the protocol.
5.2. FINAL DESIGN

The knowledge of the other values (certificate \((A, e)\) and secret \((f, v)\) values) is achieved by performing the protocol.

This protocol can be divided in the following steps:

1. The card chooses a value \(v' \in \{0, 1\}^{l_n + l_e}\), and computes the value \(U := R^n S^{v'} \mod n\). The card sends the value \(U\) to the issuer through the host. Actually we would need the EK (as in the DAA scheme) or other way for ensuring that the value \(U\) comes from a trusted card, but we do not perform any authentication and assume that the value is computed by a trusted card. The secret value \(f\) is chosen at random as specified in Sect. 5.1.2.

2. The card proves the knowledge of the secret values \((f, v')\) used to obtain the value \(U\) by performing following the signature of knowledge:

\[
SPK\{(f, v') : U \equiv \pm R^n S^{v'} \mod n \quad \land \quad f \in \{0, 1\}^{l_n + l_e + l_n + 2} \quad \land \quad v' \in \{0, 1\}^{l_n + l_e + l_n + 2} (n_t || n_i)\}.
\]

3. The issuer chooses a value \(\hat{v} \in \{0, 1\}^{l_v - 1}\) and a prime \(e \in [2^{l_v - 1}, 2^{l_v - 1} + 2^{l_v - 1}]\) and computes

\[
v'' := \hat{v} + 2^{l_v - 1} \quad \text{and} \quad A := \left(\frac{Z}{U S^{v''}}\right)^{(2^{l_v - 1})} \mod n\).
\]

Then it sends the tuple \((A, e, v'')\) to the host. The issuer has to prove the correctness of the values of this tuple:

\[
SPK\{(d) : A \equiv \pm \left(\frac{Z}{U S^{v''}}\right)^d \mod n\} (n_h)\).
\]

Note that the way of building the value \(v''\) ensures that its length is exactly \(l_v\). This means that we could choose directly \(v''\) at random as long as it has length \(l_v\).
To compute the value $A$ we have to calculate an $e^{th}$-root, i.e., we have to obtain a value $d$ such that $A := (Z_{US}^{v''}/n)^d \pmod{n}$. Such a value $d$ is called modular multiplicative inverse of $e$ with respect to the modulus $n$ (because $1/e = d \pmod{n}$) and it only exists if $e$ is coprime to $n$. To find the value $d$ we use a function that implements the Extended Euclidean Algorithm [29]. The use of this algorithm requires the knowledge of the prime factors of $n$ and $e$. This is the reason of choosing the value $e$ as prime. The prime factors of $n$ form the issuer’s secret key $SK = (p,q)$. The outputs are two integers $s$ and $d$ such that $(sn + de) \pmod{n} = \gcd(n,e)$, where $d$ is the value searched.

The value $d$ is the modular multiplicative inverse of $e$. Note that due to $\gcd(n,e) = 1$ we can write the statement $(sn + de) \pmod{n} = 1 \pmod{n}$. The associative property in the modular operations let us to write the previous state as $(sn \pmod{n}) + (de \pmod{n}) = (1 \pmod{n})$. The first term in the latter equation is zero since the modulus $n$ is a prime factor of the value $sn$, then we have that $de \pmod{n} = 1$, so $d$ is the modular multiplicative inverse of $e$.

4. The host stores the certificate $(A,e)$ and forwards $v''$ to the card.

5. The card sets $v := v' + v''$. Its secret values are $(f,v)$.

It must be taken into account that the two proofs of knowledge that this protocol requires to prove the correctness of the values, are not implemented in our system. However, such proofs of knowledge are advisable.

5.2.3 Signing Protocol

By performing the Signing protocol a user can prove that she has got a credential by proving that she has got a Camenisch-Lysyanskaya signature, and that she has the knowledge of the secret values used to compute such signature. Like in the Join protocol, all computations required to manage the pseudonyms are removed. The main difference between the Signing protocol of our final design and the protocol described in the original DAA is the number of values $T$ used in the signature of knowledge. In the original version of the DAA scheme, the values $T_1$, $T_2$ and $T'_2$ are used to commit the knowledge of the certificate and the secret values. We use only one value $T_1$ in our final design.

In the following subsections we explain in detail the computations and messages required between the three entities involved in this protocol (card, host and verifier). The messages forwarded are shown in Figure 5.2. The values on top of the objects represent the knowledge before performing this protocol.

Connection Request

In order to avoid reply attacks, the verifier generates a nonce when the user sets the connection. This nonce is linked to the connection and it is used when the verifier performs the verification algorithm.
5.2. FINAL DESIGN

This step presents no changes with respect to the original DAA specification. The nonce is computed at random with the same length as the hash function of the protocol: \( n_v \in \{0,1\}^l \).

**Signing algorithm**

Once the host receives the nonce, the host, the card and the verifier perform together a signature of knowledge. Here the user can show the verifier that she has got a credential. In detail, the user proves the knowledge of the certificate and the secret values in which such certificate is built on. Note that the certificate is kept on the host and the secret values in the card, so the proof must be performed by both entities together.

Because of the reduction in the number of parameters \( T_i \) the signature is smaller (in bits) and the computation time decreases. One can make the comparison between the following signature of knowledge and the one described in the original scheme (see Eq. 3.9). The expression of our signature of knowledge is:

\[
SPK\{(e,v,f) : ZT_1^{-2^l+1} \equiv \pm T_1^e S^v R^f (\mod n) \land f \in \{0,1\}^{l_e+l_f+l_H+2} \land e \in \{0,1\}^{l_e+l_f+l_H+2} (n_t||n_v||m) \}. \tag{5.4}
\]

Note that the equation \( ZT_1^{-2^l+1} \equiv \pm T_1^e S^v R^f (mod n) \) involves all parameters we want to prove: the certificate \((A,e)\) and the secret values \((f,v)\). The value \( A \) is present in the parameter \( Z \), since \( Z = R^f S^v A^e \), as we can see in the specification of the Join protocol. The value \( T_1 \) depends also on \( A \) as explained below. The other two equations included in the signature of knowledge prove the correct construction of values \( f \) and \( e \).

The tuple \((n_t||n_v||m)\) attached at the end of the proof, is composed by two nonces and the message \( m \) we want to authenticate. These three values are used to compute the challenge \( c \) in the host. Note that, because the message \( m \) is
used while computing the challenge $c$, potential attackers cannot intercept the proof and change the message because the verification algorithm in the side of the verifier uses that message to compute the challenge $c$ and check whether or not it matches with challenge received from the host.

The following steps, depicted in Figure 5.2, describe how to perform this proof:

1. First the host picks a random integer $w \in \{0,1\}^{l_n+l_\phi}$ and computes the value
   \[ T_1 := AS^{-w} (mod n) , \]  
   to obfuscate the value $A$ that can only be known by itself.

2. The card chooses two random integers $r_v$ and $r_f$ and computes $\tilde{T}_1$, to obfuscate the secret values $f$ and $v$, as follows:
   \[ r_v \in R \{0,1\}^{l_r+l_\phi+l_H} , \quad r_f \in R \{0,1\}^{l_r+l_\phi+l_H} , \]
   \[ \tilde{T}_1 := R^{r_v} S^{r_f} (mod n) . \]  
   The card sends $\tilde{T}_1$ to the host.

3. The host picks the random integers:
   \[ r_e \in R \{0,1\}^{l'_e+l_\phi+l_H} , \quad r_w \in R \{0,1\}^{l'_e+l_n+l_\phi+l_H} , \]
   and computes the following value:
   \[ \tilde{T}_1 := \tilde{T}_1 T_1^r S^{r_w} (mod n) . \]  
   The host is able to build the first part of the challenge:
   \[ c_h := \mathcal{H}((n||R||S||Z)||T_1||\tilde{T}_1||n_v) , \]  
   and send $c_h$ to the card.

4. The card receives the value $c_h$, chooses a random value $n_t \in \{0,1\}^{l_\phi}$ and computes:
   \[ c := \mathcal{H}(\mathcal{H}(c_h||n_t)||m) . \]  
   The card computes the $s$-values that contain the secret values:
   \[ s_v := r_v + cv , \quad s_f := r_f + cf , \]  
   and send the $s$-values, the value $n_t$ and the challenge $c$ to the host. The $s$-values are used to perform the verification algorithm as described later on.
5. The host computes the rest of the s-values required (the ones that need the knowledge of the parameters kept in the host):

\[ s_e := r_e + c \cdot (e - 2^l e - 1) , \quad s_\tilde{v} := s_v + r_w + c(e \cdot w) . \]  

(5.11)

Note that the s-value \( s_\tilde{v} \) merges the two s-values \( s_v \) and \( s_w \) used in the previous specification. This is valid since \( s_v \) and \( s_w \) are exponents of the same base \( S \) in the verification algorithm.

6. Finally the host is able to build the signature

\[ \sigma := (T_1, c, n_\ell, (s_\tilde{v}, s_f, s_e), m) . \]  

(5.12)

Note the substantial simplification of this signature with respect to the one defined in the original specification (see Eq. 3.22). In the new proof there is a reduction in the number of the commitments \( T \) (this signature has only one) as well as in the s-values used.

The host sends this signature to the verifier and the verifier performs the verification algorithm to check whether or not the signature is valid.

**Verification algorithm**

Once the verifier has the signature, it performs the verification algorithm in order to declare the signature as valid. Note that to perform this algorithm the verifier requires the public key of the issuer organization that granted the credential and the nonce sent by the verifier to the host. The main task of this algorithm is to compute the same value \( \hat{T}_1 \) as in the host, in order to build the challenge \( c \) and check whether or not it matches with the received challenge (that is included in the signature \( \sigma \)).

Therefore, the verifier computes:

\[ \hat{T}_1 := (Z T_1^{2^l e - 1})^{-c R^{s_f} T^{s_e} S^{s_v} (mod n)} , \]  

(5.13)

and checks whether:

\[ c \in \mathcal{H}(\mathcal{H}(n || R || S || Z) || T_1 || \hat{T}_1 || n_\ell || m) , \]  

(5.14)

\[ s_f \in \{0, 1\}^{l_f + l_\ell + l_n + 1} , \quad s_e \in \{0, 1\}^{l_f + l_\ell + l_n + 1} , \]  

(5.15)

all hold.
62 CHAPTER 5. DESIGNING AN ANONYMOUS CREDENTIALED SYSTEM
Chapter 6

Design of a Privacy Preserving e-Petition System

An identity management system in the digital world aims to enable users to use digital services as they use their equivalents in the physical world. For example, when Alice wants to buy tobacco or alcohol, she has to prove that she is of a certain age. For that purpose she presents her ID card, so that her age can be verified. We would like to allow her to do the same in the digital world – i.e., possess some document that she can use to prove information about herself online, without revealing more than the required information.

In this chapter we describe a case study of a digital identity management system. Our case study is based on a privacy preserving electronic petition signing system. For simplicity from now on we will refer to such a system as an "e-Petition system". The system aims to enable a user to view petitions that are published online, choose one or more of them and sign them without revealing any personally identifiable information and without any of these actions being able to be linked to each other.

Considering petitions we need to bear in mind the manner in which petitions are handled in the physical world. People are usually required to present their names and sometimes their national identity numbers. Such requirements are acceptable in the physical world because of the difficulty it would present to process this information or to steal it. In the digital world, however, stealing information is just copying it and leaving the original where it belongs and processing information is much faster than in the physical world. For this reason providing data such as names and identification numbers when signing electronic petitions should be avoided. Most e-petition systems available now ask for the signers’s name and e-mail address in order to place a signature [1, 3]. Some even
ask for a postal code [1].

The e-Petition system that we examine in this chapter makes use of the Direct Anonymous Attestation protocol proposed by Brickell et al. in [15] and described in chapter 3.1 of this paper. This way no unnecessary information is transmitted to the petition server.

6.1 Architecture

In the physical world credentials are issued by institutions to individuals and then used to prove information to other individuals or institutions in order to gain access to content or information. The e-Petition system that we describe in this chapter also consists of such three components - the Issuer component, the Verifier component and the Client component. We present each of them in more detail in the sections to follow. We start with the Issuer component (sect. 6.1.1), followed by the Verifier component (sect. 6.1.2) and conclude with the Client component (sect. 6.1.3).

6.1.1 Issuer component

In the physical world, credentials are issued to individuals after checking that these individuals are entitled to receive such credentials.

The Issuer we will be issuing credentials to every client that can perform the valid protocol. The Issuer is composed of only two parts - the Credential Issuer and the Issuer Manager, illustrated in fig. 6.1. The Credential Issuer is initialized with the public and private issuing keys and is only responsible for running the issuer part of the Direct Anonymous Attestation (DAA) protocol. The Issuer Manager is the starting point of the Issuer. It takes care of the issuing keys and of the communication that takes place between the Issuer and the Client.

6.1.2 Verifier component

The Verifier component of the e-Petition system is comprised of the two parts shown in Fig. 6.2. The Verifier Application is made up of two parts. One of them is the Petition Server, which presents information about the petitions and
6.1. ARCHITECTURE

The numbers of signatures. The second is the Verifier Manager. It maintains the communication with the Client and accepts incoming signing requests. After a request is found valid. It is also responsible for checking for double-signing attempts and updating the signature count for a given petition.

The Credential Verifier is a single entity that is contacted by the Verifier Manager when a petition signing request is received. It is responsible for checking the validity of the request and responding with a positive or negative outcome of the check.

6.1.3 Client component

The Client component of the e-Petition system has two parts: the Client Application and the Digital Wallet. Each of these two parts is in turn composed of two smaller parts (see Fig. 6.3.)

The Petition Browser and the Client Manager make up the Client Application. The Petition Browser is used by the user to view existing petitions and choose petitions for signing. When the user selects a petition, a request
for signing it is sent to the Client Manager. The Client Manager is then re-
sponsible for contacting the necessary applications that aid it in performing the
Show part of the DAA protocol. Before a request from the Petition Browser
can be handled, the Client Manager also has to aid in the completion of the
Join protocol, ensuring the possession of a valid credential by the user.

The Digital Wallet is made up of a smart card and a Host Application. The
smart card is a trusted computing device. It stores the user’s secret value and
credential and performs the all computations that include these values. The
Host Application carries out the host part of the DAA protocol. It is only
contacted by the Client Manager and submits all its answers back to it.

6.2 Architecture Requirements

The specific purpose of our system demands us to revise the requirements of a
credential system. This section presents discusses the necessary changes.

6.2.1 Security

The requirements for protection against physical and digital exposure are still
valid and therefore unchanged. Restricting multiple usage of the same credential
in the context of one petition is now mandatory and no longer optional. The
message that is signed by the credential must contain the identification of the
petition that the user wants to sign. A requirement that is introduced in the
e-Petition System is the need for universal verifiability of the signatures for each
petition.

Protection against physical exposure The master secret value of the user
and the user’s credential should be stored in a secure way, so that they
can not be extracted and shared.

Protection against digital exposure All computations involving the user
master secret value and credential should be done using a secure com-
putation device.

Multiple usage restriction A user should be able to sign multiple petitions
with one credential. However, multiple signature attempts on one petition
should be detected and stopped.

Content tampering prevention The signed petition is indicated in a mes-
sage that is signed using the anonymous credential of the user, thus proving
the user really wants to sign this petition.

Universal verifiability After a petition is signed, each user should be able to
check that her signature has been received and counted with the rest of
the signatures to the petition. The user must also be able to check that
all the saved signatures are valid signatures.
6.2.2 Privacy

The privacy requirements are still valid with the clarification that a user’s actions include not only petition signings, but also petition viewings. This is necessary because having information about what petitions a user is viewing, might lead to the identification of the user.

**Data minimization** The system should not reveal any personally identifiable information about the user.

**User activity unlinkability** Individual user actions should not be able to be connected to each other. Every action should be likely to have been performed by a user, as much as it is likely to have been performed by any other user. This includes petition viewings as well as petition signings.

6.2.3 Usability

The usability requirements are concerned with making the system more appealing to the end user. This includes making the transition from another system easy, as well as making the user comfortable and confident with using the system.

**No additional hardware requirements** The system should not require the user to acquire any additional hardware in order to be able to make use of it.

**No additional software requirements** The system should not require the user to install any additional software in order to be able to make use of it.

**Ease of transition** The system should not require any additional mental effort from the user compared to existing similar systems. No additional passwords or codes should be required to be remembered.

**Transparency of actions** The system should present clearly what actions it is performing in every given moment.
Part III

Implementation
Chapter 7

Implementation of a Privacy Preserving e-Petition System

7.1 Implementation

This section gives details on the actual implementation of each of the e-Petition Demonstrator components described in the previous chapter. We start by giving a general description of the implementation decisions that we have taken and then provide details in developing each component.

For the purpose of building this demonstration system we decided to use a software implementation of the DAA protocol provided by Sucasas [36]. This implementation assumes the simplification to the DAA protocol explained in Chapter 5, which speed up the protocol without degrading its security. We choose this implementation because it fulfils our requirements for building a proof of concept system.

One of the simplifications made in Sucasas’ implementation is that he removed the DAA pseudonyms from the protocol. These pseudonyms are, however, necessary for our e-Petition system, so we re-introduce them.

The implementation is done in the C++ programming language. We keep the core logic of the implementation. We only modify it that it can easily be split into smaller pieces, gaining flexibility. Using this modified code we developed the following applications:

KeyBuilder Called only once when the Issuer is started for the first time. It is responsible for generating the public and private issuing keys. The generated keys are saved, so that they can be used in the future as well.
CHAPTER 7. E-PETITION SYSTEM IMPLEMENTATION

CertBuilder Performs the issuer part of the computations of the Join protocol. It receives the card input for the Camenisch-Lysyanskaya signature [18] and provides as output the certificate that is to be stored by the host and the second part of the credential that is to be transmitted to the Smart card.

ServerVerifier Operates on the verifier side of the system. It is responsible for producing the nonce that the client uses as input for the proof. The application also receives and checks the proof provided by the client. It outputs a positive or negative result depending on the validity of the proof.

Prover Resides with the Client component of the Demonstrator. It participates in preparing the proof of knowledge of the anonymous credential. The application performs the host calculations from the DAA protocol. It gets input from and provides data for the Java card, until finally the zero knowledge proof for the verifier is prepared and returned.

All of these applications are written in C++ and compiled using the g++ compiler.

7.1.1 Language choice

The e-Petition Demonstrator uses an implementation of the simplified DAA protocol that was developed separately prior to the planning of the Demonstrator. This implementation is done in the C++ programming language, which must make it modular and extendable [36]. After reviewing the code of the protocol implementation, we decided that it was not suitable for usage as it is. We had to do some extensive changes to the interface provided by it. We also had to consider maintaining communication channels and user interface.

The Java programming language provides an easy and convenient way for handling communication between applications running on different hosts. It also makes creating and maintaining graphical user interfaces easy through the use of the Swing and Abstract Window Toolkit (AWT) toolkits, with which we are familiar. Since the original protocol implementation had to be changed anyway, we decided that it was best to use Java as the main programming language, and adapt the Sucasas' implementation for the purpose. The adaptation is discussed in Section 7.1.2.

7.1.2 Application layers

Because of the difference in languages used in the implementation of the protocol and of the Demonstrator itself, it makes sense to divide the implementation in several layers according to the purpose they serve. The layers that we distinguish are the computation layer, the communication layer, the application layer and the presentation layer.
7.1. IMPLEMENTATION

Computation

The computation layer consists of the updated implementation of the simplified DAA protocol. The update consists of re-factoring the classes of this implementation and developing several applications that utilize these classes. The aim is to create a set of C++ applications that can be called from the Java code when big-number computations are necessary. The applications are designed, so that they carry out only actions related to the DAA protocol. This way they are not dependent of the application of the DAA protocol. They can be used for petition signing as well as for anything else requiring anonymous credentials.

The applications that we have developed are:

Communication

The communication layer is present in all of the issuing, verifying and host applications. As its name suggests, it is responsible for maintaining the communication that takes place between the different components of the system.

Issuer The communication layer of the issuing application exchanges data with three other applications. When the Issuer is started for the first time, the issuing application calls the KeyBuilder application and receives the freshly generated public and private issuing keys. When the keys are set up communication with the host application is established and maintained. In the mean time information is exchanged with the CertBuilder application, which provides information to be sent to the host application using the established communication channel.

Verifier The verifier application communication layer is responsible for receiving the proof provided by the host application and passing it to the ServerVerifier application to be checked. The result of the check is transmitted back to the host using the same communication channel.

Host The host application is the central piece in the architecture. It delivers communication between the Issuer and the Java card during the Join protocol. Afterwards it accepts requests for building a valid proof according to parameters provided by the application layer and participates in computing the proof of knowledge of the anonymous credential stored in the card. While the proof is computed, the communication layer maintains channels to the server, the java card, the Prover application and the user browser that is to receive the outcome of the operation.

Application

The application layer is the part that is specific to the e-petition application of the system. It incorporates all the logic that is specific to e-petitions and could be replaced in order to provide a different service based on anonymous credentials.
CHAPTER 7. E-PETITION SYSTEM IMPLEMENTATION

Issuer  Since the Issuer is only concerned with providing the user with a valid credential, there is no e-petition specific logic in this component of the system.

Verifier  The application layer of the verifier component performs several very important functions in the e-Petition Demonstrator system. This layer consists of two components - the web server that runs on the verifier side and the verifier application.

The web server is used for providing easy access to users to the e-petition data. This data includes petitions available for signing together with their full description, number of signatures for each petition and, most importantly, a list of all the valid proof that have been received by the verifier application. This list allows each user to verify that their signature has been recorded along with all other valid signatures.

The verifier application’s application layer is responsible for providing the content of a petition that is requested by the communication layer. It is also responsible for checking for double-signing attempts by the users. If such an attempt is not detected, and the received proof is found to be valid, it is the application layer that increases the signature count for the given petition and that saves the received proof for future reference.

Host  The application layer of the host application is the component that awaits a petition signing request from the user browser. After receiving such a request it directs the communication layer to fetch the petition body, which is then presented to the user for verification. Following an approval from the user, the layer signals the communication layer that a credential show must be conducted. After the result from the credential show is received, the application layer is responsible for transmitting it back to the user’s browser as well as to the presentation layer, so that the user is properly informed of the outcome of the action.

Presentation

Since the e-Petition Demonstrator system is designed as a proof of concept, the Issuer and the Verifier have no graphical user interface. The presentation layer is only present in the host application and serves for providing the user with information about the current stage of the communication that takes place between the host application and all the other applications. The host presentation layer is also used to show the user the body of the petition they will be signing and to get their approval as a security measure against data tampering within the user’s browser. The presentation layer is also used as a means for obtaining the user’s PIN code, which is supposed to provide access to the functionality of the java card.
Chapter 8

Server and Issuer Software Implementation

This chapter contains the software specification of our implementation. We describe how using our interface we build a program to obtain and show a credential. A more comprehensive explanation can be found in [36]

In Sect. 8.1, we show the interface implemented and a suitable way of making use of its functions. We give descriptions of all the functions and illustrate through examples how they can be used. In Sect. 8.2 we offer a more detailed explanations of the implemented software. The main decisions taken are presented in this section, as well as several extensions of the functionality of the interface.

Our interface provides the functions required by a host to be able to obtain and show an anonymous credential. Thus, the main entity of our system is the host, that performs the protocol by being in touch with a real (or simulated) card, issuer and verifier. When we say simulated entity we refer that the interaction with the card the issuer or the verifier can be simulated if this entities are not available (e.g., when evaluating the implementation). However, in order to prove the functionality of our protocol in a better approach to a realistic scenario, we have the issuer and the verifier as real entities (independent processes running in the same or other computer), and an example of the card implementation is described in Chapter 9. However, we note that the use of such implementations are not required for making use of our interface and that it could be used together with any other implementations of such entities.
8.1 Program Design

The main steps in the protocol are the obtaining by the host of a credential and the showing of this credential to a verifier. These steps, as in the mathematical design presented in Sect. 5.2 result in the Join protocol and the Show protocol (also called Signing protocol). Our first approach to an implementation consists in these two steps together with an initialization phase in which the communication is set and all variables are initialized.

In Fig. 8.1 the initialization of the protocol is shown, where the host starts the communication with the issuer and obtains the issuer’s public key. As we can see in the figure the host obtains the key from the issuer and sends it to the card and the verifier. The name of the functions and the values returned are over the arrows, which are ordered according the order in which the host executes them. Although it would be possible to send the public key from the host to the verifier in this step, in the DAA protocol the key is sent at the same time as the credential. Note that this way getting a credential and its showing are independent. Otherwise, due to the fact that the verifier can only validate a credential with respect to the public key of the issuer organization that issued that credential, if the host sends the issuer’s public key to the verifier at the beginning of the protocol she would only be able of showing credentials obtained from the issuer owner of the public key previously sent. As an example of this, a user could obtain e-cash (credentials that represent money) from two different banks and use them after getting them both, otherwise she would have to obtain the first credential, show it, and afterwards get the other one.

In a second step the host performs the Join protocol together with the card and the issuer. In Fig. 8.2 we show the steps in the Join protocol, the name of the functions in which this protocol is splitted and the values returned are over the arrows, which are ordered according the order in which they are executed by the host. These steps are implemented in different functions in our interface.

After performing these two steps, the host and the card have the values whose knowledge is the requirement to successfully carry out the Show protocol. Figure 8.3 depicts the different steps done by the host to perform this protocol, the name of the functions in which this protocol is splitted and the values
8.1. PROGRAM DESIGN

Figure 8.2: Join protocol splitted in functions.

returned are over the arrows, which are ordered according the order in which
the host executes them. Note that the values shared between the card and the
host are hidden under the function \texttt{build\_proof()}, that outputs the signature $\sigma$
(see Eq. 5.12), called proof, that the host sends to the verifier. Note also that
the public key is sent together with the proof.

Figure 8.3: Show protocol splitted in functions.

In order to construct a modular and extendable interface, composed by the
functions depicted in the previous figures, we have chosen an object oriented
programming language, C++. Thus, we define the different classes in which we
wrap the functions mentioned. We must take into account that the issuer, the
card and the verifier could not be implemented (as happens while evaluating), so
our interface must provide functions to simulate these entities. Thus, besides the
design of classes to hold and construct values as the public key of the issuer, the
certificate and the signature, we need classes to manage the (real or simulated)
communications with the issuer, the card and the verifier.

Figure 8.4 gives an overview of the design of the project. In this figure we
assume that we have real entities, this means the issuer the verifier and the card
are independent processes in contact with the host. The different C++ classes
used in each process are presented next to the entities. As an example, the entity
verifier, placed in the main file “server.cc”, makes use of the classes \texttt{PKI} and
\texttt{Proof}. The host uses the classes \texttt{PKI, Proof, Prover} and \texttt{Certificate} to handle
all the parameters needed in the protocol. The classes \texttt{CARD}, \texttt{ISSUER} and \texttt{SERVER}, also placed in the main file “client.cc”, manage the communications with the other entities as shown in the figure. These three classes are named this way because in case of the three entities with which they are in touch are not present (issuer, verifier and smartcard), case of the evaluation, they can make all the computations by themselves in order to obtain the parameters that they should receive from such entities. This way the host does not realize whether or not the entities are really present (actually, the presence of real entities must be notified to the host in the file client.cc while calling the constructors of each class as we can see in Sect. 8.1.5, 8.1.7 and 8.1.5 respectively). The functionality of each class is summarized in table 8.1

Table 8.1: Classes implemented in the interface

<table>
<thead>
<tr>
<th>Class</th>
<th>Functionality</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prover</td>
<td>performs computations in the host side</td>
</tr>
<tr>
<td>Proof</td>
<td>stores the signature (proof) sent during the Show protocol</td>
</tr>
<tr>
<td>Certificate</td>
<td>stores the certificate obtained in the Join protocol</td>
</tr>
<tr>
<td>PKI</td>
<td>generates and stores the issuer’s public and secret key</td>
</tr>
<tr>
<td>SERVER</td>
<td>manages the communications/simulation with/of the verifier</td>
</tr>
<tr>
<td>ISSUER</td>
<td>manages the communications/simulation with/of the issuer</td>
</tr>
<tr>
<td>CARD</td>
<td>manages the communications/simulation with/of the smartcard</td>
</tr>
</tbody>
</table>

Note that despite in this text we say that the card has the issuer’s public key we do not include the class PKI in the entity CARD depicted in Fig. 8.4. This is because the card does not use the four values of the public key, but only three, thus, we have chosen to pass these three values in three separated variables to it.

If we assume that the issuer, verifier and card are not present, we can execute the protocol with only one process, the host, as shown in Fig. 8.5, note the difference with Fig. 8.4. In this case the classes \texttt{CARD}, \texttt{ISSUER} and \texttt{SERVER} make the computations as stated before. Note that we could have only one or two simulated entities, in the sense that we could have a real issuer but not a card or a verifier. This can be chosen in the main file “client.cc”.

In the following subsections we describe each class and its interface. Note that all variables we describe are from the C++ type “ZZ”. This type is provided by the library “ntl”. It must be taken in account that due to the big numbers handled by our scheme we must use “ZZ” integers instead of the “long” integers provided by C++. The latter keeps the information in only 4 bytes, establishing an upper bound of $2^{32}$ for numbers that could require more than two thousands bits, $2^{2048}$ (see the suggested lengths in the original DAA protocol in Sect. 3.3). We only describe the functions that compose the interface, the rest of internal
Figure 8.4: General overview of the classes used by the entities. Different processes are inside circles. The classes used in each process are inside rectangles.

functions can be found in the source file “classes.hh”.

### 8.1.1 Class PKI

An instance of this class allows the construction of the issuer’s secret and public key. As shown in Figure 8.6 the attributes of this class are the values belonging to the $PK_I = (n, R, S, Z)$ and the $SK = (p, q)$. The constructor does not generate these values, which are only initialized to zero. The generation is performed by the function `build_values()`.

Note that all the values, even the ones that belong to the secret key are declared as public. This makes easier the management of the values in all the computations along the protocol.

In our system all entities have an object PKI (except the card) but only the issuer is allowed to call the function `build_values()`. The rest of the entities obtain the key from the issuer, as shown in Figure 8.1 in the case of the host, and in Figure 8.3 in the case of the verifier. To prevent the rest of the entities from obtaining the secret key, the copy constructor (`PKI(const &PKI)`) and the equal operator (`PKI& operator= (const PKI&)`) are built in such a way that the variables $p$ and $q$ (i.e., the secret key) are set to zero when we try to copy an object “key” (in order to copy an object we can only use the equal operator or the copy constructor).

The following piece of code shows an example of how the issuer can generate
the secret and public key and share only the public key. Note that this code is only an example and it is not used in our program.

```c
ISSUER issuer; // instance of class ISSUER
PKI key // instance of class PKI

issuer.key.build_values(); // in issuer.key is the SK and the PK
key = issuer.key; // in key is only the PK
```

In this example, if later we want to obtain the secret key we must include the following instructions:

```c
key.p = issuer.key.p
key.q = issuer.key.q
```

### 8.1.2 Class certificate

This object stores the values of the certificate \((A, e)\), that are obtained by performing the Join protocol. In Figure 8.7 shows the definition (attributes and functions) of this class. Note that this object does not perform any computation, its functionality is only the storage of the mentioned values.

### 8.1.3 Class Proof

In order to perform the not interactive proof of knowledge carried out in the Show protocol, the host and the card compute a signature \(\sigma\) that is sent to
8.1. PROGRAM DESIGN

8.1.4 Class Prover

The class Prover provides all the computations and the values required to show a credential. The values that this class has as attributes are used to compute the signature, and most of them are also part of it. The main function in this class is build_proof(), that outputs a proof using the certificate, the nonce received from the verifier and the issuer’s public key. This function also manages the communication with the card, whose secret values must be used to compute the proof. Figure 8.9 depicts this class.
8.1.5 Class CARD

The host performs the communication with the card using the interface provided by this object. Furthermore, in case of not having a real card this object performs all the computations by itself, simulating the card’s behavior. Figure 8.10 shows the definition of this class.

The function \texttt{build} \_value \_U() is used to obtain the commitment on the secret value \( f \) stored in the card in order to obtain the C-L signature from the issuer (see the Join Protocol in Sect. 5.2.2). After obtaining the C-L signature the host uses the function \texttt{build} \_value \_vpp() to set the secret value \( v \) in the card. The function \texttt{ask} \_for \_values() and the “get methods” (functions to obtain values that can be seen in the source file \texttt{classes.hh}) that this class provides are used by the function \texttt{build} \_proof() of the class Prover, thus, these functions can be considered as part of the interface, but actually users of our interface do not use them directly.

The constructor of this class is overloaded in such a way that we can choose between a simulation and a real communication by including the number 1 or 0 respectively as an input. As an example:

\begin{verbatim}
CARD card; // real communications
CARD card(0); // real communications
CARD card(1); // simulation
\end{verbatim}

The function \texttt{set} \_communication() sends the issuer’s public key to the card and, in case of a real entity is present, establishes the communications. It
must be taken into account that the real communications with the card are not provided yet. How to implement this communication remains as future work.

### 8.1.6 Class SERVER

In order to perform the Show protocol, the host must obtain a nonce $n_v$ from the verifier and send the signature $\sigma$ to such verifier. To manage these communications we have the class SERVER. We chose the name server assuming that in a real scenario the verifier would run in a server, provided by the service provider of the system towards whom the user wants to authenticate herself. However, a more suitable name for this class would be verifier. This class is depicted in Fig. 8.11.

In this class we have defined again an overloaded constructor in such a way that we can choose between simulation and real communications as the following example shows:

```cpp
SERVER verifier;  // real communications
SERVER verifier(0);  // real communications
SERVER verifier(1);  // simulation
```
Note that in this example the name of the class is SERVER but we have chosen the more suitable name “verifier” for the object (this object is an instance of the class SERVER).

The function `set_communications()` establishes the communications with the server. In order to perform the Show protocol, the host can obtain the nonce from the verifier with the function `ask_for_a_nonce()`. The host can send the credential and the public key of the issuer organization that commits the validity of the certificate used to build the credential with the function `send_proof()`. This function returns the result of the verification algorithm performed by the verifier.

### 8.1.7 Class ISSUER

The class ISSUER provides all the functionality required to perform the Join protocol. This class provides a function `set_communication()` that not only establishes the communication, but also returns the issuer’s public key, as shown in Fig. 8.12. The function `build_certificate()` returns the C-L signature of the secret value stored in the card. It is called “build certificate” because the main values that this function outputs are the tuple $(A, e)$ (see Join protocol in Sect. 5.2.2).

Like in the previous classes, we can choose between real and simulated communications as the following example shows:
8.2 Code Specification

This code specification consists in a formal description about the interface implemented at a user's level. In this section we describe how to use the functions provided by the classes implemented, such classes are shown in table 8.1.

8.2.1 List of functions

In this section we only describe the high-level functions of our implementation. With high-level functions we refer to the functions that provide the functionality of the protocol designed. Thus, the following set of functions allows a host to perform the protocol in order to get and show a credential.
int check_security_parameters();
This function checks whether or not the values of the security parameters in
the file security_parameters.hh fulfill the equations as described in the math
specification. Concretely, the two equations that $l_v$ and $l_e$ must hold with
respect to the other values.

void PKI::build_values();
This function generates the values that belong to the public and secret keys
according the equations mentioned in the math specification.

int Proof::verify_signature(PKI& key, ZZ& nv);
This function is called by an object proof with the nonce and the public key
as inputs. It returns the result of the verification algorithm. If the algorithm
succeeds returns 1, otherwise returns 0.

Proof Prover::build_proof(CARD&, ZZ& nonce, PKI&, certificate&);
This function builds the signature $\sigma$ by performing the computations described
in the Show protocol. This signature (proof) is computed by the host together
with the card using the knowledge of the certificate stored in the host and the
secret values kept in the card. The first input of this function is the object card
because the communications with this entity are wrapped by this function. The
other inputs are the nonce received from the verifier, the issuer’s public key and
the object certificate that stores the tuple $(A,e)$. The value outputted is an
object Proof.

void SERVER::set_communication(string s, int port);
This function sets the communication by sockets with the server (verifier). The
first input “string s” is the ip address of the computer in which the verifier is
running. As an example, in an evaluation scenario where the server is running
in the same computer as the host, that input must be “localhost”. The second
input is the port where the server is running. In case of having a simulated
server none inputs are used.

ZZ SERVER::ask_for_a_nonce();
With this function the host sends a request to the server asking for a nonce to
begin the Show protocol. It has no inputs and it outputs the nonce.

int SERVER::send_signature(PKI& ,Proof&);
With this function the host sends the signature (an object Proof) and the
issuer’s public key to the server. It returns true or false according to the result
of the verification algorithm. In case the verification succeeds this function
returns 1, otherwise it returns 0.

void CARD::set_communication(ZZ R, ZZ S, ZZ n);
This function sets the communication with the smartcard and sends the values
$R$, $S$ and $n$ belonging to the issuer’s public key. It must be taken into ac-
count that the real communications with the card are not provided yet, so this
function gives the public values to the object that simulates the smartcard.
8.2. CODE SPECIFICATION

`ZZ CARD::build_CapU();`

This function returns the commitment computed by the card on the secret value \( f \), i.e., \( U = R / S v \mod n \). See Join protocol in Sect. 5.2.2.

`void CARD::build_value_v(ZZ vpp);`

This function has as input the value \( v'' \) (received from the server). It computes the secret value \( v = v' + v'' \) in the card. See Join protocol in Sect. 5.2.2.

`PKI_ISSUER::set_communication(string s, int port);`

This function sets the communication by sockets with the issuer and returns the issuer’s public key. The first input “string s” is the ip address of the computer in which the issuer is running, as an example, if we run the server in the same computer as the host, that input must be “localhost”. The second input is the port where the issuer is running. In case of having a simulated issuer none inputs are used.

`void ISSUER::build_certificate(ZZ U, ZZ& A, ZZ& e, ZZ& vpp);`

This function receives as input the value \( U \) (commitment received from the card), and outputs the values \( A, e \) and \( vpp \). Thus, with this function the host obtains the C-L signature, i.e., the certificate \( (A, e) \) and the value \( v'' \) that must be forwarded to the card.
Chapter 9

Smart Card

Implementation

This chapter describes the implementation of the simplified version of the DAA protocol as presented in chapter 5. As is clear from the protocol description, several complex mathematical operations are necessary for the implementation. However, as explained in chapter 4, Java Card platforms do not provide a possibility for low-level programming to allow code optimisation, and the Java virtual machine causes a lot of overhead. It is therefore not desirable to implement complex algorithms in Java when efficiency is crucial. The execution of the Java code for these would be far too slow. On the other hand, the Java Card standard includes an API, which allows for certain cryptographic operations to be executed very efficiently on a dedicated coprocessor, or by using an efficient cryptographic library. In section 9.1 we start by giving an overview of the complete functionality our applet should provide, the problems this poses for an implementation on Java Card platforms, and the tools we use for the development. Sections 9.2, 9.3 and 9.4 then describe the actual implementation, while section 9.5 provides details about the timings achieved and the memory consumption for the applet. Details about functions implemented and APDU commands used to communicate with the applet are given in section 9.6.

9.1 Overview

For the implementation of the simplified DAA protocols, we need several complex mathematical operations. This section presents an overview of the functionality needed, the problems we face when implementing this functionality on Java Card platforms, and the tools we used for the implementation.
9.1.1 Functionality Description

The Java Card applet discussed here has to implement the TPM part of the simplified DAA protocol as presented in chapter 5. This protocol consists of two sub-protocols, namely the join protocol and the signing protocol. Figure 9.1 shows a complete overview of the functionality our applet has to provide.

![Diagram](image)

Figure 9.1: Overview of the functionality provided by the Java Card DAA applet.

9.1.2 Problems

The basic problem for the implementation of cryptographic protocols like the DAA protocol is that they work with large numbers (which means in this case 1024 bits and more). Support for basic calculations – such as addition, multiplication, etc. – with these big numbers is not part of the Java Card 2.1.1 and 2.2.1 standards, which means we have to implement a BigNum library, providing all the necessary mathematical support for the protocol. Table 9.1 shows an overview of the mathematical operations needed for the DAA protocol.

A second difficulty when implementing a BigNum library on Java Card platforms, or smart cards in general, is memory limitation. Like all smart cards,
Table 9.1: Mathematical operations for the simplified DAA protocol.

<table>
<thead>
<tr>
<th>Mathematical operation</th>
<th>Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Random number generation</td>
<td>} directly available on the cards</td>
</tr>
<tr>
<td>Hash (SHA-1)</td>
<td></td>
</tr>
<tr>
<td>Large number addition</td>
<td>see section 9.2</td>
</tr>
<tr>
<td>Large number multiplication</td>
<td>see section 9.4</td>
</tr>
<tr>
<td>Modular multi-exponentiation</td>
<td>see section 9.3</td>
</tr>
</tbody>
</table>

Java Card smart cards only have a very limited amount of persistent as well as transient memory. When implementing a cryptographic protocol like DAA where complex mathematical operations need to be executed on very large numbers, storing these numbers can cause problems. To keep calculations fast enough, we should limit the amount of intermediate results stored in EEPROM, and instead use the faster (but smaller) RAM.

9.1.3 Development Tools

Several tools exist that help in developing Java Card applets. Sun provides the Java Card Technology Development Kit [8], which contains:

- A Java Card compiler, outputting CAP, EXP and JCA files. The JCA file (Java Card Assembly) is a human-readable ASCII version of the CAP file for testing and debugging.
- A simulator, to test applets before uploading them to an actual card.
- Sample source code and documentation.

This toolkit does not provide a graphical interface itself, but an open source tool (EclipseJCDE, [9]) exists to integrate the toolkit into the Eclipse development environment [2].

For testing the implementations of the algorithms presented in this chapter, we also need a tool to generate test values, and calculate the correct results for comparison with the implementation’s output. Magma is a software package designed to solve computationally hard problems in algebra, number theory, etc. [13]. It supports e.g. random number generation, random prime generation, modular exponentiation, etc.

9.1.4 Platforms

The implementation described in this chapter was tested on two different Java Card platforms. Some general specifications for these platforms are:

- **Card 1**: Java Card 2.1.1, GlobalPlatform 2.0.1, 64K EEPROM
Card 2: Java Card 2.2.1, GlobalPlatform 2.1.1, 72K EEPROM.

The GlobalPlatform Card Specifications define a secure and dynamic card and application management system [4]. They specify card components, command sets, etc.

9.2 Large Number Addition and Subtraction

The two most basic operations for a BigNum library are addition and subtraction. This section presents algorithms for basic integer as well as modular addition and subtraction.

Large Integer Addition and Subtraction

Algorithms 1 and 2 describe how two large numbers can be added, respectively subtracted. They are based on the classical pencil-and-paper methods and derived from the algorithms presented in [30], but extended to allow for numbers of different size to be added (subtracted). Additions are done word by word, starting from the least significant position. For each addition a carry bit has to be handled, which means that the final result will be at most one bit longer than the largest operand. Therefore one extra word has to be reserved in memory.

Algorithm 1 Large number addition

Require: \( x = (x_{len1-1}, \ldots, x_0)_b \) and \( y = (y_{len2-1}, \ldots, y_0)_b \)
Ensure: \( z = x + y = (z_{outLen-1}, \ldots, z_0)_b \)

1: \( w := 0; \) \{the length of \( w \) must be at least on bit more than the word size\}
2: \( outLen = \max(len1, len2) + 1; \)
3: \( \text{for } i = 0 \text{ to } outLen - 1 \text{ do} \)
4: \( \text{if } i < len1 \text{ then} \)
5: \( w := w + x_i; \)
6: \( \text{end if} \)
7: \( \text{if } i < len2 \text{ then} \)
8: \( w := w + y_i; \)
9: \( \text{end if} \)
10: \( z_i := w(0..\text{wordLength}-1); \)
11: \( w := w(\text{wordLength}); \) \{carry bit\}
12: \( \text{end for} \)
13: \( z_{outLen-1} := w; \)
Algorithm 2 Large number subtraction

**Require:** \( x = (x_{len1-1}, \ldots, x_0)_b \) and \( y = (y_{len2-1}, \ldots, y_0)_b \)

**Ensure:** \( z = x - y = (z_{outLen-1}, \ldots, z_0)_b \)

1: \( w := 0; \)
2: for \( i = 0 \) to \( len1 - 1 \) do
3: \( w := w + x_i; \)
4: if \( i < len2 \) then
5: \( w := w - y_i; \)
6: end if
7: if \( w < 0 \) then
8: \( z_i := w + b; \)
9: \( w := -1; \)
10: else
11: \( z_i := w; \)
12: \( w := 0; \)
13: end if
14: end for

Modular Addition and Subtraction

The algorithms described above can be easily extended to their respective modular operations. These extensions are described in algorithms 3 and 4. These algorithms assume, however, that the inputs are smaller than the modulus \( m \).

Algorithm 3 Large number modular addition

**Require:** \( m = (m_{len1-1}, \ldots, m_0)_b, x = (x_{len1-1}, \ldots, x_0)_b \) and \( y = (y_{len2-1}, \ldots, y_0)_b, (x, y < m) \)

**Ensure:** \( z = x - y \mod m = (z_{outLen-1}, \ldots, z_0)_b \)

1: \( z := x + y; \) \{algorithm 1\}
2: if \( z > m \) then
3: \( z := z - m; \) \{algorithm 2\}
4: end if

9.3 Modular Multi-Exponentiation

The most complex operation needed for the DAA protocol implementation is exponentiation.

**Definition 9.1.** The *modular exponentiation* of base \( b \) to the power \( e \) modulo
Algorithm 4 Large number modular subtraction

Require: $m = (m_{len-1}, \cdots, m_0)_b$, $x = (x_{len1-1}, \cdots, x_0)_b$ and $y = (y_{len2-1}, \cdots, y_0)_b$, $(x, y < m)$

Ensure: $z = x - y \mod m = (z_{outLen-1}, \cdots, z_0)_b$

1: $z := x - y$; \{algorithm 2\}
2: if $z < 0$ then
3: $z := z + m$; \{algorithm 1\}
4: end if

$m$ is represented as $b^c \mod m$.

Definition 9.2. The modular multi-exponentiation of bases $b_1$ and $b_2$ to the powers $e_1$ and $e_2$ modulo $m$ is represented as $b_1^{e_1} \cdot b_2^{e_2} \mod m$.

When computing a multi-exponentiation, it is in general inefficient to compute the powers $b_i^{e_i}$ separately and then multiply them. Instead, efficient multi-exponentiation algorithms are used. For example, algorithms exist that combine the bases $b_i$ during a precomputation stage, and then process all exponents simultaneously during the actual evaluation stage. For more information on these algorithms we refer to [31].

Although these algorithms can calculate a multi-exponentiation efficiently, their implementation in Java would still be far too slow, and the Java Card API does not provide any support for exponentiation. However, the API does include support for RSA, as described in section 4.2.4. In this section we describe how we can use the RSA functionality to calculate modular (multi-) exponentiations.

9.3.1 RSA and Exponentiation

The RSA scheme is a public-key cryptosystem that can be used to provide confidentiality as well as digital signatures. The public RSA key is represented by $(n, e)$, the secret key is $d$. Encryption and decryption work as follows:

- Encryption of message $m$ to ciphertext $c$: $c = m^e \mod m$
- Decryption of ciphertext $c$ to message $m$: $m = c^d \mod m$

Details can be found for example in [30].

It is already clear from these equations that calculating a modular exponentiation is not much different from executing an RSA encryption or decryption, provided it is possible to set all the values involved (base, exponent, and modulus).

9.3.2 RSA and Java Card

Possible keyType values for RSA on Java Card are TYPE_RSA_CRT_PRIVATE, TYPE_RSA_PRIVATE and TYPE_RSA_PUBLIC. Using one of these values as the keyType
9.3. MODULAR MULTI-EXponentiation

will generate either an RSAPrivateCrtKey, RSAPublicKey or RSAPrivateKey
object. On the last two, the methods setModulus(...) and setExponent(...) are available. The different modes of operation for RSA on Java Card platforms
are:

- ALG_RSA_ISO14888
- ALG_RSA_ISO9796
- ALG_RSA_NOPAD
- ALG_RSA_PKCS1

The difference between these is merely the way input values are padded, i.e.
what is added to the message before encryption, respectively removed from the
decryption result.

Using the Crypto Hardware for Simple Modular Exponentiations

A general modular exponentiation algorithm needs to allow for the base, exponent,
and the modulus to be set. This means that the implementation of the
RSA algorithm on the card is not allowed to add any data that is not explicitly
given to it. Instead the RSA exponentiation has to be done with the exact data
provided. Requirements for the RSA API are thus:

1. The ability to set the values message (base), e (exponent) and n (modu-
lus) if the encryption mode is used, respectively ciphertext, d, and n for
decryption mode. These values will of course not necessarily satisfy the
RSA equations.

2. The used RSA method should not add any other data to these values
before executing the actual exponentiation.

The RSA key objects that satisfy requirement 1 are TYPE_RSA_PRIVATE and
TYPE_RSA_PUBLIC. The object TYPE_RSA_CRT_PRIVATE generates an RSAPrivateCrtKey,
which will use the Chinese Remainder Theorem (CRT). For decryption with the
CRT the prime number p and q from which the modulus n is derived are needed,
see [30]. Therefore this mode is not suited for our case, where the factorisation of
n is not known. Requirement 2 is satisfied only by the mode ALG_RSA_NOPAD. All
other modes use a certain padding mechanism which possibly adds some data to
the message (base). This is no problem if RSA is actually used for encryption,
because the padding can then simply be removed again upon decryption, but
makes the modes useless if the value we are interested in is the result of the
exponentiation. When calculating modular exponentiations, the used values for
exponent and modulus will not necessarily satisfy the RSA equations, i.e. they
are not part of a valid public/private key pair. Because the setExponent and
setModulus methods just store the given values, testing is needed to determine
if this causes problems.
Tests

To make sure the exponentiation actually works, testing needs to be performed. Two things need to be considered:

- Does the exponentiation execute at all? This means, does the command execute without throwing an exception?
- If the code returns the status word 0x9000 (OK), is the returned result also correct?

To check the correctness of the implementation, we used Magma to generate random test values and to calculate the correct results. We used these values as input to a Java Card simulator, as well as real cards. The moduli we used were prime numbers and RSA moduli (i.e. the product of two prime numbers). All tests were performed on a 512-bit (64-byte) RSA algorithm. Our results are shown below:

- For all tests where the modulus was the full 64 bytes long (i.e. the most significant byte not set to zero), no exceptions or incorrect exponentiation results occurred.
- The real Java Cards never threw exceptions, and always returned a result.
- When the most significant byte of the modulus is set to 0x00, results become implementation dependent.
  - The simulator crashes
  - Some cards give the correct result.
  - Other cards give the correct result, but in a shifted version. This is however still usable.

We did not perform extensive tests on non-prime and non-RSA moduli, because this is not needed for the implementation of the DAA protocol. We know however that it is e.g. not possible to use an even number as the modulus.

Extension for Long Exponents

One of the exponentiations in the DAA protocol uses an exponent longer than the modulus. The RSA functions provided by the crypto API only accept exponents with sizes up to the size of the modulus, so it is not possible to execute this complete exponentiation directly with the API. In this section we explain how exponentiations with exponents longer than the modulus can be calculated.

As before, the result of the exponentiation \( b^e \mod n \) is needed. The difference, however, is that the exponent \( e \) is now larger than the modulus \( n \). Since
9.3. MODULAR MULTI-EXponentiation

it is not possible anymore to feed this arbitrary-length \( e \) directly to the RSA hardware we split the exponent as follows:

\[
e = e_l \parallel e_{l-1} \parallel \ldots \parallel e_2 \parallel e_1
\]

\[
\begin{align*}
e_1 &= e[0 \rightarrow \text{rsa.length} - 2] \\
e_2 &= e[\text{rsa.length} - 1 \rightarrow 2 \cdot (\text{rsa.length} - 1)] \\
\vdots \\
e_l &= e[(l - 1) \cdot (\text{rsa.length} - 1) + 1 \rightarrow \text{length}(e) - 1].
\end{align*}
\] (9.1)

We can now write the exponentiation \( b^e \mod n \) as \( b^{e_l \parallel e_{l-1} \parallel \ldots \parallel e_2 \parallel e_1} \mod n \).

Now, for each of the \( e_i \) values, a modular exponentiation is executed to produce the intermediate results \( b^{e_i} \mod n \). To combine the intermediate results to the result of the actual exponentiation, we rewrite the splitting of the exponent as follows:

\[
e = e_l \parallel e_{l-1} \parallel \ldots \parallel e_2 \parallel e_1
\]

\[
\begin{align*}
e_1 &= e_1 + e_2 \cdot 2^{(\text{rsa.length} - 1)} + \ldots + e_l \cdot 2^{(l - 1) \cdot (\text{rsa.length} - 1)} \]
\] (9.2)

To keep equations simple, in the following we only explain the case where the total exponent fits in two sub-exponents \( e_1 \) and \( e_2 \). This is also the maximum needed for the implementation of the DAA protocol, since both cards we use support RSA up to 1024 bits. We thus write the exponentiation as:

\[
b^e \mod n = (b^{e_2 \cdot 2^{(\text{rsa.length} - 1)} + e_1}) \mod n
\]

\[
= [(b^{e_2 \cdot 2^{(\text{rsa.length} - 1)}} \mod n) \cdot (b^{e_1} \mod n)] \mod n.
\] (9.3)

The first exponentiation in equation 9.3 still has an exponent that is too large to calculate directly with the crypto API. Splitting is however straightforward:

\[
b^{e_2 \cdot 2^{(\text{rsa.length} - 1)}} \mod n = (b^{e_2} \mod n)^{2^{(\text{rsa.length} - 1)}} \mod n.
\] (9.4)

All exponents involved now have at most the size of the modulus, so all the exponentiations can be calculated using the RSA hardware. As a result of all the transformations described above, we converted one modular exponentiation involving an exponent with size larger than \( \text{rsa.length} \) into several modular exponentiations and multiplications. For example, when \( e \) is composed of only two sub-exponents, three modular exponentiations and one modular multiplication have to be calculated. Clearly, the problem of multi-exponentiation remains. The next section discusses how we can solve this.

9.3.3 Modular Multi-Exponentiation

The previous sections described how single modular exponentiations can be computed using RSA hardware available on Java Card platforms. However,
for the DAA protocol we need modular multi-exponentiations. As explained in the introduction to this section, clever algorithms exist to calculate modular multi-exponentiations. None of these algorithms uses single modular exponentiations as a basic operation to calculate a multi-exponentiation though. Therefore it is not possible to use the RSA functionality when using a clever multi-exponentiation algorithm. This would make the implementation too slow, so we use the RSA functionality in the crypto API to calculate single exponentiations, and multiply the intermediate results. Clearly, we still need a modular multiplication algorithm to do this.

9.4 Large Number Multiplication

9.4.1 Large Integer Multiplication

Large number multiplication is one of the most-used operations in cryptographic protocols. Algorithm 5, based on the algorithm in [30], describes a simple large number multiplication. The algorithm is based on the classical pencil-and-paper method shown in figure 9.2. Multiplication is done word by word, creating a carry word each time. Therefore, the algorithm uses a double word for the storage of intermediate results. The total size of the product is the sum of the sizes of the operands.

Figure 9.2: Classical pencil-and-paper row-wise multiplication method.
Algorithm 5 Large number multiplication

Require: $x = (x_{len1-1}, \cdots, x_0)_b$ and $y = (y_{len2-1}, \cdots, y_0)_b$

Ensure: $z = x \cdot y = (z_{len1+len2-1}, \cdots, z_0)_b$

1: $z := (0, 0, \cdots, 0)_b$
2: $carry := 0$
3: for $i = 0$ to $len_1 - 1$
4:    $carry := 0$
5:    for $j = 0$ to $len_2 - 1$
6:       $(w_1 w_0)_b := x_j \cdot y_i + z_{i+j} + carry$
7:       $z_{i+j} := w_0$
8:       $carry := w_1$
9:    end for
10: $z_{i+len2} := w_1$
11: end for

9.4.2 Modular Multiplication

Since the product of two equally sized large numbers can be double this size, a straightforward extension to modular multiplication, as was possible for addition and subtraction, is not possible here. Several standard solutions exist though:

- Special modular multiplication algorithms
- ‘Naive’ method: Interleaved row-multiplication and reduction

Special algorithms such as Montgomery multiplication become advantageous only when many modular multiplications have to be executed. For example, Montgomery multiplication works by first converting the numbers to be multiplied to the Montgomery domain, multiplying there in a more efficient way, and in the end converting back from the Montgomery domain. These conversions create too much overhead when just a single multiplication is needed, so Montgomery multiplication is not a suitable solution in this case. Integer multiplication with Barrett reduction has the same problem, plus the fact that it needs more memory to store the intermediate multiplication result.

The basic interleaved row-multiplication and reduction algorithm is algorithm 6. This algorithm is based on the interleaved modular multiplication algorithm described in [26].

The problem with this algorithm however is that it does not yet specify how to calculate line 3. The algorithm needs to know how many times it has to subtract the modulus from the intermediate result for this result to become smaller than the modulus. To calculate lines 3 and 4, again several possibilities exist:

- Calculate the division in line 3. Division is the most time consuming of all
Algorithm 6 Interleaved row-multiplication and reduction

Require: \( m = (m_{len-1}, \ldots, m_0)_b \), \( x = (x_{len-1}, \ldots, x_0)_b \) and \( y = (y_{len-1}, \ldots, y_0)_b \)

Ensure: \( z = x \cdot y \mod m \)

1: for \( i = 0 \) to \( len - 1 \) do
2: \( z := z + y_i \cdot x \cdot b^i \);
3: \( q := \left\lfloor \frac{z}{m} \right\rfloor \);
4: \( z := z - q \cdot m \);
5: end for

basic mathematical operations, making this not an option for Java Card platforms.

- While-loop: keep subtracting and comparing the intermediate result with the modulus until it is smaller. This will still be very slow because of the large amount of subtractions and comparisons needed.

- Estimate the number of times the modulus has to be subtracted. A Barrett-like estimating technique can be used for this. This method is explained below.

A method as described in [25] can be used to estimate the number of times the modulus needs to be subtracted from the intermediate result. This estimator will never overestimate, so the subtraction will not result in a negative number. It is now possible to rewrite lines 3 and 4 of algorithm 6, leading to algorithm 7.

Algorithm 7 Interleaved row-multiplication with fast reduction

Require: \( m = (m_{len-1}, \ldots, m_0)_b \), \( x = (x_{len-1}, \ldots, x_0)_b \) and \( y = (y_{len-1}, \ldots, y_0)_b \), \( \mu = MS_2(b^{len}/m) \) \{MS2 denotes the selection of the two most significant bits\}

Ensure: \( z = x \cdot y \mod m \)

1: for \( i = 0 \) to \( len - 1 \) do
2: \( z := z + y_i \cdot x \cdot b^i \);
3: \( q := MS_2(MS_2(z) \cdot \mu) \);
4: \( z := z - q \cdot m \);
5: end for

In our particular scenario where fast modular exponentiations are available (see section 9.3), a third option becomes interesting. We can convert modular multiplications to modular exponentiations, which can be calculated with the
9.4. LARGE NUMBER MULTIPLICATION

fast cryptographic coprocessor. Equation 9.5 shows two ways to do this.

\[
\begin{align*}
(1) \quad & (a + b)^2 - a^2 - b^2 = 2 \cdot a \cdot b \pmod{m} \\
(2) \quad & (a + b)^2 - (a - b)^2 = 4 \cdot a \cdot b \pmod{m}
\end{align*}
\] (9.5)

It is clear that these equations allow to obtain the result of a multiplication without actually calculating this multiplication. However, all the operations that are still present have to be calculated modulo \( m \). The following operations are required:

- modular addition (algorithm 3)
- modular subtraction (algorithm 4)
- modular exponentiation (see section 9.3)
- **modular division by 2 or 4**: modular right shift (by one or two positions)

All modular operations, except for modular right shifting, were already described in previous sections. Algorithm 8 presents a modular right shifting algorithm. Table 9.2 compares the number of operations needed for both possibilities in equation 9.5. Depending on the speed of the cryptographic hardware on specific cards, it is possible to decide which one will be the fastest.

**Algorithm 8** 1-bit modular right shifting

**Require:** \( m = (m_{len-1} \cdots m_0)_b, x = (x_{len-1} \cdots, x_0), (x < m, m \text{ odd}) \)

**Ensure:** \( z = x >> 1 \bmod{m} \)

1: if \( x_0(0) == 1 \) then
2: \( x := x + m; \)
3: end if
4: for \( i = 0 \) to \( len - 1 \) do
5: \( z_i := x_{i+1}(0)||x_i(\text{end..1}); \)
6: end for

Table 9.2: Comparison of the number of operations needed for the different possibilities for modular multiplication presented in equation 9.5

<table>
<thead>
<tr>
<th>Operation</th>
<th>number of executions</th>
<th>(1)</th>
<th>(2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>modular addition/subtraction</td>
<td></td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>modular exponentiation</td>
<td></td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>1 bit modular right shift</td>
<td></td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>
This fast multiplication mechanism can also be used for normal (non-modular) multiplication, by applying algorithm 5 with a larger word size. The basic multiplications in the algorithm are then the modular multiplications described above. As long as the word size is kept below $\sqrt{l_m}$, where $l_m$ is the length of the modulus, the modular multiplications have the same result as regular multiplications.

9.4.3 Code Optimisation

Although it is not possible to do low-level programming for Java Card platforms, it is still possible to use the Java bytecode for code optimisation. By carefully analysing bytecodes provided in JCA files by the Java Card compiler and comparing them to the Java code, it is possible to determine what kind of Java instructions produce more or less bytecode instructions. To clarify, we give some examples of possible optimisations:

- Single-assignment should be used as much as possible.
- Conditional statements should be avoided.
- Loops with increasing index are preferable over decreasing index, since an `inc` bytecode instruction exists.

9.4.4 Memory Management

Because the transient memory (RAM) on smart cards is limited, different methods have to be applied than for programming regular Java. Because garbage collection is not present, all memory should be allocated upon initialisation. RAM has to be shared by different functions requiring transient memory for storing intermediate results. We solve this by declaring a temporary byte array for which memory is reserved just once upon applet instantiation. A reference to this array is then passed to every method requiring temporary memory space, together with the starting point in the array from where it is still free.

9.5 Results

9.5.1 Timing Details

Table 9.3 presents an overview of the (on-card) execution times of the protocol. The times include communication to and from the cards, but not the execution times on the host. We did tests on the two platforms mentioned in section 9.1.4, and took average timings of 15 executions.

We believe that the reasons for faster execution on card 2 is related to the fact that this is a newer card, with probably a more efficient virtual machine implementation and cryptographic coprocessor.
### 9.6. IMPLEMENTATION DETAILS

#### 9.6.1 List of Functions

This section provides an overview of the functions present in the DAA applet, with short descriptions of their purpose. Firstly, the following is a list of the

#### 9.5.2 Memory Consumption

Table 9.4 shows the memory consumption of our applet, making clear that code size nor volatile memory consumption pose a problem for the platforms we use. Note that the code size includes only the size of the applet’s own code, not APIs it might use which are already present on the cards. Also, the RAM consumption does not include the RAM consumption caused by the use of the built-in RSA methods, since memory for these is reserved in a different way by the crypto API itself.

<table>
<thead>
<tr>
<th>Memory type</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>EEPROM (code size)</td>
<td>3648 bytes</td>
</tr>
<tr>
<td>EEPROM (variables)</td>
<td>2227 bytes</td>
</tr>
<tr>
<td>RAM</td>
<td>605 bytes</td>
</tr>
</tbody>
</table>

Table 9.3: Execution times for the different parts of the DAA applet.

<table>
<thead>
<tr>
<th>Part of the protocol</th>
<th>Card 1</th>
<th>Card 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Startup (connect card and select applet)</td>
<td>0.1 sec</td>
<td>0.1 sec</td>
</tr>
<tr>
<td>Join SPK request</td>
<td>13.1 sec</td>
<td>2.3 sec</td>
</tr>
<tr>
<td>Set credential</td>
<td>0.5 sec</td>
<td>2.7 sec</td>
</tr>
<tr>
<td><strong>Total join protocol</strong></td>
<td><strong>13.6 sec</strong></td>
<td><strong>5.0 sec</strong></td>
</tr>
<tr>
<td>Sign SPK request</td>
<td>14.0 sec</td>
<td>3.5 sec</td>
</tr>
<tr>
<td>Hash request</td>
<td>0.2 sec</td>
<td>0.1 sec</td>
</tr>
<tr>
<td>Get $s_v$</td>
<td>8.5 sec</td>
<td>1.5 sec</td>
</tr>
<tr>
<td>Get $s_f$</td>
<td>2.4 sec</td>
<td>0.3 sec</td>
</tr>
<tr>
<td><strong>Total signing protocol</strong></td>
<td><strong>25.1 sec</strong></td>
<td><strong>5.4 sec</strong></td>
</tr>
</tbody>
</table>
high-level functions, that are directly called by command APDUs. (for more details about the different protocol steps, see chapter 5)

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>public static void install(byte[] bArray, short bOffset, byte bLength)</td>
<td>After uploading the applet code (through the .cap file, see chapter 4), this method installs the applet. The install method creates a new instance of the DaaApplet class, thus calling its constructor.</td>
</tr>
<tr>
<td>public DaaApplet()</td>
<td>This is the constructor of the DaaApplet class. Tasks of the constructor are to initialise persistent and transient memory variables, thereby reserving memory for them for the whole applet lifetime. It also initialises the necessary cryptographic routines, and calls the random data generator to set the secret key value $f$.</td>
</tr>
<tr>
<td>public void process(APDU apdu)</td>
<td>This is the command dispatcher of the applet. Based on the value of the INS byte, it sends the command to one of the top-level processing methods. (see chapter 4 for more information about APDU commands)</td>
</tr>
<tr>
<td>private void setKeyValue(APDU apdu)</td>
<td>This method provides the possibility to set the issuers public key values, when the applet is not locked. For testing and/or debugging purposes, it is also possible to set the secret key values $f$ and $v$.</td>
</tr>
<tr>
<td>private void getKeyValue(APDU apdu)</td>
<td>For debugging and testing purposes, this method provides the possibility to get the issuers public key values, as well as the card’s secret key values, out of the card (when the card is not in locked state). Note that for a real application, methods for getting and setting secret keys must be deleted.</td>
</tr>
<tr>
<td>private void lock(APDU apdu)</td>
<td>After calling this method, it is not possible anymore to set any of the key values. To unlock the applet, the applet has to be reinstalled.</td>
</tr>
<tr>
<td>private void joinRequest(APDU apdu)</td>
<td>Join request (the first step of the join protocol)</td>
</tr>
<tr>
<td>private void setCredential(APDU apdu)</td>
<td>The second step of the join protocol, resulting in the new secret key value $v$ being set.</td>
</tr>
<tr>
<td>private void spkRequest(APDU apdu)</td>
<td>SPK request (the first step in the signing protocol)</td>
</tr>
<tr>
<td>private void hashRequest_0(APDU apdu)</td>
<td>Hash request (the second step in the signing protocol)</td>
</tr>
<tr>
<td>private void hashRequest_1(APDU apdu)</td>
<td>Calculate and return $s_v$</td>
</tr>
<tr>
<td>private void hashRequest_2(APDU apdu)</td>
<td>Calculate and return $s_f$</td>
</tr>
</tbody>
</table>

The reason the hash request step is split into three parts is that the APDU
buffer is too short to accommodate for all the result data to be sent back at once. Therefore, multiple requests have to be sent to the card to retrieve the complete response data.

Most of the functions shown above need additional lower-level functions providing mainly the basic and modular multi-precision arithmetic functionality explained in this chapter. A list of those functions is shown below:

```java
private short add(byte[] src1, short src1Off, short length1, byte[] src2, short src2Off, short length2, byte[] dest, short destOff):
  Addition of two BigNum byte arrays.

private short addEq(byte[] src1, short src1Off, short length1, byte[] src2, short src2Off, short length2):
  Addition of two BigNum byte arrays, where the result is written in the location of the first byte array. (operation src1 += src2)

private void addMultiple(byte[] src1, short src1Off, byte[] src2, short src2Off, short length, byte multiple, byte[] dest, short destOff):
  Addition of a byte-multiple of src2 to src1

private short multiply(byte[] src1, short src1Off, short length1, byte[] src2, short src2Off, short length2, byte[] dest, short destOff):
  Multiplication of two big number byte arrays.

private void subtractEq(byte[] src1, short src1Off, short length1, byte[] src2, short src2Off, short length2):
  Subtraction of two BigNum byte arrays, where the result is written in the location of the first byte array. (operation src1 -= src2)

private boolean compare(byte[] src1, short src1Off, byte[] src2, short src2Off, short length):
  Compare two byte arrays.

private short expModN(byte[] base, short baseOffset, byte[] exp, short expOffset, byte[] mod, short modOffset, byte[] outBuff, short outOffset):
  Modular exponentiation with setting of the modulus. The setting of the modulus is only necessary the first time an exponentiation is calculated.

private short expMod(byte[] base, short baseOffset, byte[] exp, short expOffset, byte[] outBuff, short outOffset):
  Modular exponentiation without setting the modulus. The modulus previously set through expModN(...) is used.

private void multiply2(byte[] src1, short src1Off, short length1, byte[] src2, short src2Off, short length2, byte[] dest, short destOff, short tempOff):
  Multiplication of two BigNum byte arrays, using method (1) in equation 9.5.
```
private void modMul(byte[] src1, short src1Off, byte[] src2, short src2Off, byte[] src3, short src3Off, short length, byte[] dest, short destOff, short tempOff): Modular multiplication of two big number byte arrays, where the method used to calculate this is: 
\((a + b)^2 - a^2 - b^2) >> 1 == a \times b\)

private void shiftRight(byte[] src, short srcOff, short length): Shift right by one bit.

9.6.2 APDU Commands

All high-level protocol functions are called with their respective APDU commands, through the command dispatching process method. Table 9.7 lists the different APDU commands.

<table>
<thead>
<tr>
<th>CLA</th>
<th>INS</th>
<th>P1</th>
<th>P2</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>01</td>
<td>00</td>
<td>00</td>
<td>Join request</td>
</tr>
<tr>
<td>00</td>
<td>02</td>
<td>00</td>
<td>00</td>
<td>set credential</td>
</tr>
<tr>
<td>00</td>
<td>10</td>
<td>00</td>
<td>00</td>
<td>SPK request</td>
</tr>
<tr>
<td>00</td>
<td>20</td>
<td>00</td>
<td>00</td>
<td>Hash request</td>
</tr>
<tr>
<td>00</td>
<td>21</td>
<td>00</td>
<td>00</td>
<td>get sf</td>
</tr>
<tr>
<td>00</td>
<td>22</td>
<td>00</td>
<td>00</td>
<td>get sv</td>
</tr>
<tr>
<td>F1</td>
<td>01</td>
<td>00</td>
<td>R</td>
<td>set R</td>
</tr>
<tr>
<td>F1</td>
<td>02</td>
<td>00</td>
<td>S</td>
<td>set S</td>
</tr>
<tr>
<td>F1</td>
<td>03</td>
<td>00</td>
<td>n</td>
<td>set n</td>
</tr>
<tr>
<td>F1</td>
<td>11</td>
<td>00</td>
<td>f</td>
<td>set f*</td>
</tr>
<tr>
<td>F1</td>
<td>12</td>
<td>00</td>
<td>v</td>
<td>set v*</td>
</tr>
<tr>
<td>F2</td>
<td>01</td>
<td>00</td>
<td>–</td>
<td>get R</td>
</tr>
<tr>
<td>F2</td>
<td>02</td>
<td>00</td>
<td>–</td>
<td>get S</td>
</tr>
<tr>
<td>F2</td>
<td>03</td>
<td>00</td>
<td>–</td>
<td>get n</td>
</tr>
<tr>
<td>F2</td>
<td>11</td>
<td>00</td>
<td>–</td>
<td>get f*</td>
</tr>
<tr>
<td>F2</td>
<td>12</td>
<td>00</td>
<td>–</td>
<td>get v*</td>
</tr>
<tr>
<td>FF</td>
<td>00</td>
<td>00</td>
<td>–</td>
<td>lock the key values</td>
</tr>
</tbody>
</table>

* these commands must be deleted
Chapter 10

Conclusions

In this report we have described the work that we have done on e-government in the scope of the ADAPID project. In our previous report we have highlighted the privacy implications of using unique identifiers to access services in which sensitive personal information has to be revealed [22].

Current Belgian electronic ID cards use X.509 public key certificates and digital signatures to authenticate their holders. These certificates contain unique identifiers that allow service providers to link all transactions of a citizen, limiting the level of privacy that can be achieved. In our previous report we built a demonstrator that proves that the current Belgian e-ID can be used to bootstrap advanced identity management methods, and provide privacy in e-government services.

In the second half of the project, we went a step further and explored the possibilities of an electronic ID card able to perform anonymous authentication. For this purpose, we implemented anonymous credentials on a off-the-shelf smart card with an integrated cryptographic co-processor to aid in the heavy computations required by our protocols.

The first part of this report provides some background on the basic technologies that our protocols use (anonymous credential protocols and zero-knowledge proofs), and introduces the Direct Anonymous Attestation protocol. This protocol was selected by the Trusted Computing Group [10] as standard to perform remote anonymous authentication between a trusted hardware module (as our smart card) and a verifier. We also present the smart card, as well as the Java Card standard, we used in our demonstrator.

In the second part we present a design of an Anonymous Credential System using this building blocks. To illustrate the functionality of such a system, we have developed a secure and privacy enhanced electronic petition proof-of-concept demonstrator based on our design. In particular, our application protects citizens privacy by allowing them to anonymously sign petitions using their electronic ID card, while preventing malicious users from signing a petition multiple times.
The third part details the implementation of each of the elements of our demonstrator: the credential issuer, the verifier (e-petition web server) and the client. We have described the software implementation of our application, and illustrated through examples how they can be used. We have also specified how to implement the functionalities needed by the Direct Anonymous Attestation protocol on a smart card. Finally, we have studied the performance of the signing and verifying protocols, and show that they are suitable for its use in real-world e-government applications.

Although future research is still needed to address unresolve legal and technical considerations, the results presented in this report open a door to the development of strong privacy enhancing e-government applications. We have shown that, if future electronic ID cards are equipped with low cost hardware that enable them to perform heavy cryptographic operations, they can be used by citizens to perform secure online transactions while protecting their privacy.
Bibliography


