Integrating Anonymous Signature Schemes into PKI using Dynamic Cryptographic Accumulators

Hendrik Dettmer

March 23, 2009

Diplomarbeit

Advised by: Dipl.-Ing. Sven Schäge

Department of Electrical Engineering and Information Sciences
Ruhr-University Bochum
Chair for Network and Data Security (NDS)
Prof. Dr. Jörg Schwenk

Eidesstattliche Erklärung

Hiermit versichere ich, dass ich meine Diplomarbeit selbst verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt, sowie Zitate kenntlich gemacht habe.

Bochum Deutschland, 23.03.2009
Ort, Datum

Hendrik Dettmer

Contents

1 Introduction **1**
- 1.1 Motivation . . . 1
- 1.2 Aim of the Thesis . . . 2
- 1.3 Organization of the Thesis . . . 3

2 Mathematical Background **4**
- 2.1 Notation and Definitions . . . 4
- 2.2 Elliptic Curves . . . 8
- 2.3 Zero-knowledge Protocols . . . 11
  - 2.3.1 Non-interactive Zero-knowledge Protocols . . . 13
- 2.4 Complexity Assumptions . . . 14

3 Authentication **16**
- 3.1 Digital Signature Schemes . . . 18
- 3.2 Digital Certificates . . . 19
- 3.3 Public Key Infrastructures . . . 21
  - 3.3.1 Drawbacks of PKIs . . . 21
- 3.4 Group Signatures Schemes . . . 22
  - 3.4.1 Formal Definition of Group Signatures . . . 23
  - 3.4.2 Overview of the Presented Group Signature Scheme . . . 25
  - 3.4.3 Security Requirements . . . 25

4 Dynamic Cryptographic Accumulators **29**
- 4.1 Definitions of Cryptographic Accumulators . . . 30
- 4.2 Definition of Dynamic Cryptographic Accumulators . . . 32

5 Dynamic Accumulator Scheme from Bilinear Pairings **35**
- 5.1 Features of the Scheme . . . 35
- 5.2 Overview of the Scheme . . . 36

5.3 Group Key Generation 39
5.4 User PKI Certificate Generation 39
5.5 Join Protocol 40
5.5.1 Proof of Knowledge 42
5.6 Signature Protocol 43
5.6.1 Sign 44
5.6.2 Verify 45
5.6.3 Security Proof of the Underlying Zero-knowledge Protocol 46
5.7 Update Witness Algorithm 50
5.8 Update Accumulator Algorithm 51
5.9 Open Algorithm 51
5.10 Judge Algorithm 52
5.10.1 Proof of the Non-interactive Zero-knowledge Protocol . . 53
5.11 Revoke Algorithm 55
5.12 Proof of the Security Requirements 55

6 Environment and Development **59**
6.1 Programming Language and Libraries 60
6.2 Implementation of the Scheme 61
6.2.1 User PKI Certificate Generation 61
6.2.2 Group Key Generation 63
6.2.3 Join Protocol 64
6.2.4 Signature Protocol 67
6.2.5 Update Witness Algorithm 68
6.2.6 Update Accumulator Algorithm 68
6.2.7 Open Algorithm 69
6.2.8 Revoke Algorithm 69
6.2.9 Judge Algorithm 70
6.3 Compiling on Different Platforms 70
6.4 Coding Problems 72
6.5 Integrating a Test Portal 73
6.6 Graphical User Interface 74

7 Future Work **76**

8 Conclusion **77**

A Installation Procedure **82**
A.1 Linux Installation 82
A.1.1 Server Installation 82
A.2 Firefox Plugin Installation 83

B Space Comparison with PKI-based Certificates **85**
B.1 X.509 Certificate and Signature 85
B.2 Dynamic Cryptographic Accumulator Signature 87

C Speed Comparison with PKI-based Certificates **90**

List of Figures

2.1 Example elliptic curves 8
2.2 Point addition on a elliptic curve 9
2.3 Point doubling on a elliptic curve 9

5.1 Illustration of the different use symbols 37
5.2 Illustration of a chronology overview of the scheme 38
5.3 Illustration of the group key generation 39
5.4 Illustration of the user PKI certificate generation 40
5.5 Illustration of the join protocol 40
5.6 Illustration of the sign algorithm 44
5.7 Illustration of the verify algorithm 45
5.8 Diagram of the simulation in the random oracle model 50
5.9 Illustration of the update witness algorithm 50
5.10 Illustration of the update accumulator algorithm 51
5.11 Illustration of the open algorithm 52
5.12 Illustration of the judge algorithm 53
5.13 Illustration of the revoke algorithm 55

6.1 Overview of the used programs 60
6.2 Raw description of the program CMake 71
6.3 Overview of the Mozilla plugin scheme 75

Abstract

This thesis centers around the integration of an implementation of an efficient group signature scheme with efficient revocation into a public key infrastructure. The better part of the document, the used dynamic cryptographic accumulator scheme is described. The various backgrounds of this scheme are explained and a complete approach to the issue is given. Also the practical implementation of this theoretical idea is illustrated and advices are given.
Group signature schemes allow a member of a group to anonymously sign messages on the group's behalf. In case of dispute it is possible in our scheme that a group manager can open a signature and revoke the anonymity requirement. We concentrate on the group signature scheme by Lan Nguyen that was presented at the RSA conference 2005.
The thesis contains a brief introduction to the topic together with the required mathematical and cryptographic preliminaries. We give a detailed description of our implementation and compare its space and speed complexity with existing PKI-based protocols.
Although our implementation accounts for privacy issues which are not addressed by classical PKI-based authentication, the efficiency of our solution is still comparable.

Keywords: dynamic cryptographic accumulators, elliptic curve cryptography, group signatures, bilinear pairings

1 Introduction

In this section we present the motivation for the thesis and an introduction to the field of anonymous authentication and group signatures.

1.1 Motivation

Before the invention of computers, cryptography was used mostly for confidentiality. The aim was the encryption of communication. Today, due to the connection of computers and the establishment of the Internet, many other security requirements come into focus of science.
For example, everybody who uses the Internet hopes that certain security goals hold, like the secure retrieval of E-mails over an untrustworthy network. But especially in those large networks new problems arise. Take a user who wants to buy a product in an online shop. It must be ensured that the user is connected to the correct server and not to some adversary, which is spoofing the servers address or is just exploiting a typing error of the user when entering the server's internet address.
To encounter this problem, digital certificates are used to identify the website. These certificates are sent by the server to the client and the client uses a database, embedded in the browser, to verify the certificate. Although this method has flaws, in practice it works most of the time very well.
Most users do not have an own digital certificate. So there is no way that a user can sign a contract or a bid in an online auction. Today, most of the online shops use a username and password to authenticate the user. If an adversary can find out or steal this combination, it can do everything in an online shop on the user's behalf. The user can try to deny the actions took by the adversary, but it is hard to prove that he is not responsible for them. The main problem is that most users use very short and weak passwords, which are easy to guess for an adversary.
Most of the time, the adversary does not even have to try all possible passwords. An easier way is to put up an Internet forum or any other online community and try to bait the users to this site. Most users log in on different sites with the same password, so they have lesser passwords to remember. The adversary takes the voluntarily given passwords and usernames and tries them out on many other sites.
Besides authentication another important security requirement is privacy or anonymity. In most applications anonymity is not intended because the provider

of a service wants to know which user to charge. But, as will be shown in this thesis, accountability and anonymity can both be accomplished at the same time in a cryptosystem.

1.2 Aim of the Thesis

This thesis aims at implementing a group signature scheme. A group signature scheme combines different security requirements such as anonymity and traceability. It seems that it is not possible to trace someone if everyone in the group is anonymous. However, it turns out that this intuition is not right. Through the use of zero-knowledge proofs a signer can prove that a certain value, his identification number, is in fact accumulated in a publicly accessible accumulator that is managed by a trusted third party, the so called group manager. When the group signature scheme is correctly setup, this also proves that he is a member of the group.
This solution has many advantages when compared with existing applications. For example, the computational size of the proposed scheme is not connected to the number of members in the group.
As the title of the thesis indicates, our group signature scheme is integrated into the public key infrastructure (PKI) and does not stand alone. A PKI provides the identification of entities in a network and many PKIs are very well established in practical systems. We can use these established trust hierarchies for authentication purposes and do not need to build up our own. Besides, a PKI provides certificates and they can be used to generate a secure channel, so that a new member can securely join the group.
Another aim is to provide proof-of-concept of practicality of authentication systems that account for privacy issues. We address the current trend in cryptography to base cryptosystems on elliptic curve groups with a bilinear pairing. Such systems have the advantage that cryptographic parameters can be much shorter than in traditional cryptographic settings.
Although by now there exists a large amount of literature on cryptosystems that use bilinear pairings, practical systems are rare. This is partly due to the more involved theory behind pairing-based cryptography.
All mathematical proofs and theoretical approaches are included in this thesis, to show the entire way from a mathematical idea to an implemented program. Technically, the focus of this implementation lies on the programming language C++ and the executability of the program on different platforms and operating systems.

1.3 Organization of the Thesis

This thesis is organized as follows:
Chapter 2 provides the general mathematical background. After some basic definitions, Section 2.2 presents elliptic curves. This is followed by an introduction into zero-knowledge protocols in Section 2.3. The chapter is concluded in Section 2.4, where complexity assumptions are described, which several cryptosystems are based on.
Chapter 3 discusses authentication in general and signature based authentication in more detail. The first section presents an introduction to digital signature schemes. In the next section, digital certificates and their connection to digital signatures are described. This evolves to public key infrastructures in Section 3.3. The focus then shifts on group signature schemes in Section 3.4.
Chapter 4 focuses on dynamic cryptographic accumulator schemes. After first presenting the accumulator in Section 4.1, the dynamic accumulator definition is introduced in Section 4.2. The dynamic accumulator is used in our group signature scheme for efficient revocation of group members.
Chapter 5 discusses our group signature scheme. In the first section, some features of our scheme are highlighted. After that, Section 5.2 gives an overview over the scheme. The next sections describe the different components of our scheme. The chapter concludes with Section 5.12, a proof of security of our scheme.
Chapter 6 presents the implementation of our group signature scheme. After a short overview of the developed programs, Section 6.1 describes our claim for a programming language and the used software libraries. Section 6.2 is dealing with the implementation of the programs and processes. This is followed by Section 6.3 and a discussion on compiling on different platforms. Section 6.4 discusses problems, which were encountered during the implementation phase. Then, the focus shifts on constructing a web portal for testing purposes. The chapter concludes with a presentation of a graphical user interface for our implementation in Section 6.6.
Chapter 7 discusses future work and Chapter 8 gives a conclusion.
In Appendix A the installation of our scheme is described. Appendix B compares the size of standard digital signature with the signatures generated in our scheme. In Appendix C, the executing speed of our scheme is compared to standard public key infrastructure scheme.

2 | Mathematical Background

Here, we provide the mathematical definitions required throughout this thesis. Most of the definitions are directed at introducing elliptic curves and pairings. Both play a very important role in the implemented group signature scheme.

2.1 Notation and Definitions

A finite group is a group with finitely many elements. All other definitions in this chapter make use of groups or fields to structure more complex entities.

Definition 2.1.1. *Finite Group*
A group $(\mathbb{G}, \circ)$ *is a finite set* $\mathbb{G}$ *together with a binary operation* $\circ$ *(the group operation) if both satisfy the following properties:*

- ***Closure.*** *For all* $a, b \in \mathbb{G}$ *the result of the operation* $\circ$ *yields to an element* c *also in* $\mathbb{G}$*:* $c = a \circ b$ *with* $c \in \mathbb{G}$*.*
- ***Associativity.*** *The binary group operation is associative:* $a \circ (b \circ c) = (a \circ b) \circ c, \forall a, b, c \in \mathbb{G}$*.*
- ***Identity.*** *There is an identity element* x *such that:* $a \circ x = x \circ a = a, \forall a \in \mathbb{G}$*.*
- ***Inverse.*** *For every element* $a \in \mathbb{G}$ *there must be an inverse* a^{-1} *such that:* $a \circ a^{-1} = 1 \ \forall a \in \mathbb{G}$*.*

A cyclic group is a special group in which all elements can be represented as a power of a generator of the group. In cryptography most groups are cyclic.

Definition 2.1.2. *Cyclic Group*
A group $\mathbb{G}$ *is called cyclic if it can be generated by a single element:* $\langle a \rangle = \mathbb{G}$*,* $a \in \mathbb{G}$*.*

In a multiplicative cyclic group exponentiation is easy but the computation of the discrete logarithm is believed to be very hard if the group size is sufficiently large.

Definition 2.1.3. *Discrete Logarithm*
Let p be a prime and $\mathbb{G} = (\mathbb{Z}/p\mathbb{Z})$ a cyclic group with order p. Let g be a generator of $\mathbb{G}$, so that every element in $\mathbb{G}$ can be written as $a = g^b$. For a group element x the discrete logarithm of x to the base g is an integer n such that $x = g^n$ and $n \forall n;\ 0 \leq n \leq (p-2)$.

Since elliptic curves are defined over a fields, we also present a definition of finite fields. A field is a group with some assumptions as defined below.

Definition 2.1.4. *Field*
A field $\mathbb{K}$ is defined as a finite group in which two binary operations, called addition and multiplication, are present, such that:

- *$\mathbb{K}$ is an Abelian group under addition,*
- *multiplication is associative and possess an identity element,*
- *multiplication is commutative,*
- *every nonzero element is invertible with respect of multiplication,*
- *multiplication is distributive with respect to addition.*

Central to the definition of pairings is the notion of elliptic curves, which are smooth, projective algebraic curves of genus one.

Definition 2.1.5. *Elliptic Curve*
An elliptic curve E over a field $\mathbb{K}$ denoted by $E/\mathbb{K}$ is given by the Weierstraß equation

$$E : y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6$$

where the coefficients a_1, a_2, a_3, a_4, a_6 are elements of $\mathbb{K}$.

Definition 2.1.6. *Bilinear Pairings*
Let $\mathbb{G}_1$,$\mathbb{G}_2$ be cyclic additive groups generated by points on an elliptic curve P_1 and P_2, respectively, whose orders are a prime p, and $\mathbb{G}_T$ be a cyclic multiplicative group of order p. Suppose there is an isomorphism $\psi : \mathbb{G}_2 \to \mathbb{G}_1$ such that $\psi(P_2) = P_1$. Let $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ be a bilinear pairing with the following properties:

1. ***Bilinearity*** *$e(aP, bQ) = e(P, Q)^{ab}$ for all $P \in \mathbb{G}_1, Q \in \mathbb{G}_2, a, b \in \mathbb{Z}_r$*
2. ***Non-degeneracy*** *$e(P_1, P_2) \neq 1$*

3. ***Computability*** *There is an efficient algorithm to compute* $e(P, Q)$ *for all* $P \in \mathbb{G}_1, Q \in \mathbb{G}_2$.

For simplicity and better computability, we hereafter restrict ourselves to the case $\mathbb{G}_1 = \mathbb{G}_2$ *and* $P_1 = P_2$. *The implemented scheme can easily be modified for an asymmetric pairing where* $\mathbb{G}_1 \neq \mathbb{G}_2$. *For a group* $\mathbb{G}$ *of prime order let* $\mathbb{G}^*$ *denote* $\mathbb{G}^* = \mathbb{G}\backslash\{\mathcal{O}\}$ *where* $\mathcal{O}$ *is the identity element of group* $\mathbb{G}$.

The group key generation algorithm generates the elliptic curve, the groups $\mathbb{G}_1$ and $\mathbb{G}_T$ a generator P for group $\mathbb{G}_1$ and the bilinear pairing e.

To sign long messages to an arbitrary-sized bit string must be mapped to a given fixed bit size. This can be established by collision resistant hash functions.

Definition 2.1.7. *Hash Function*
A hash function $\mathcal{H}$ *is an efficiently evaluable mapping* $\mathcal{H}$*:*

$$\mathcal{H} : \{0,1\}^* \rightarrow \{0,1\}^n.$$

$\mathcal{H}$ *maps arbitrary-sized messages to fixed-sized hash values. The integer* n *is called the output length of* $\mathcal{H}$. *The image* $\mathcal{H}(X)$ *of* X *is called the hash value of* X.

It should be very difficult to find two bit strings X_1 and X_2 which are mapped to the same output value $\mathcal{H}(X)$.

For some proofs the definition of Turing machines is needed. A Turing machine is an abstract computation model. The model was first introduced by Alan Turing 1936. A standard Turing machine consists of a tape, a head, a state register and an action table.

Definition 2.1.8. *Turing Machine*
Hopcroft and Ullman formally defined in 1979 [25] a (one-tape) Turing machine as a 7-tuple $M = (Q, \Gamma, b, \Sigma, \rho, q_0, F)$ *where*

- Q *is a finite set of states*
- Γ *is a finite set of the tape alphabet*
- $b \in \Gamma$ *is the blank symbol (the only symbol which can occur infinitely on a tape)*
- $\Sigma \subseteq \Gamma \setminus \{b\}$ *is the set of input symbols*
- $\rho : Q \times \Gamma \rightarrow \Gamma \times Q \times \{L, R\}$ *is the transition function, where* L *is a left shift,* R *is a right shift. If a* k*-tape Turing machine is defined, a third state* S *for no shift is also needed.*
- $q_0 \in Q$ *is the initial state*

- *$F \subseteq Q$ is the set of final states*

A Turing machine is deterministic if the action table has no more than one entry for a combination of symbol and state. There exist also non-deterministic machines which have more than one entry on the action table for one combination.

2.2 Elliptic Curves

This section deals with the differences between normal arithmetic functions and their counterparts on elliptic curves. Mathematical functions like addition and multiplication are defined differently in an elliptic curve group.
Through the equation $y^2 = x^3 + ax + b$, which is a simple description of an elliptic curve, it is clear that every x coordinate has two y values, $+y$ and $-y$. Every element of the elliptic curve group $E(\mathbb{F}_{p^k})$ is represented by $[x, y]$ with $x, y \in \mathbb{F}_{p^k}$ and satisfies the equation given above.
Also, the equation $2^2a^3 + 3^3b^2$ must be unequal 0, so that a, b are suitable for cryptographic purposes. As shown in Fig. 2.1, elliptic curves can have different shapes. For all demonstrative purposes in this section the curve in the middle of the figure, generated with the equation $y^2 = x^3 - x + 1$, is taken.

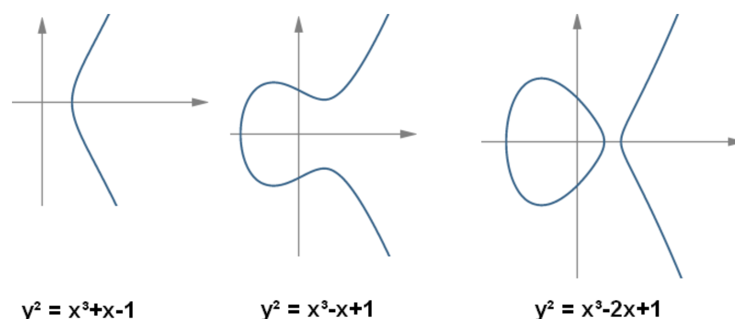


Figure 2.1: Example elliptic curves

To be able to work with points on an elliptic curve, we must define the corresponding group operations: point addition, point doubling and scalar multiplication.
For demonstrative purposes we use figures to describe the operations in a graphical way. All operations are applied to points on the elliptic curve, therefor a curve is drawn in a two-dimensional coordinate system and the operations will be executed exemplarily on this curve. The Weierstraß equation of the curve is $y^2 = x^3 - x + 1$.

Point Addition Let $P_1, P_2 \in E(\mathbb{F}_{p^k})$ be two distinct points on the elliptic curve. The result of the addition $Q = P_1 + P_2$, where $Q \in E(\mathbb{F}_{p^k})$, is graphically defined as follows: first draw a line through P_1 and P_2, when this line crosses the elliptic curve again, the point $Q^{'}$ is found. The reflection over the x-axis of $Q^{'}$ is the point Q and the sum of $P_1 + P_2$, see Figure 2.2.

Point Doubling Let $P \in E(\mathbb{F}_{p^k})$ be a point on the curve E. The result when doubling this point $Q = 2 \cdot P,\ Q \in E(\mathbb{F}_{p^k})$ can graphically be derived by

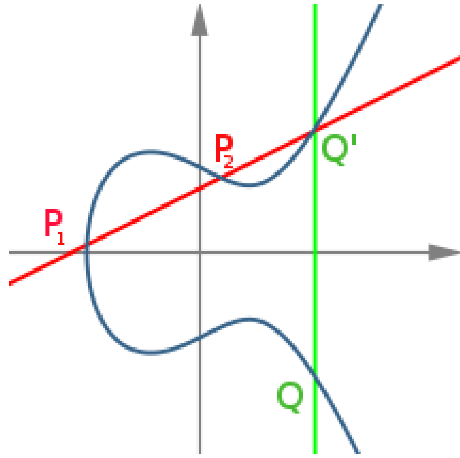


Figure 2.2: Point addition on a elliptic curve

drawing a tangent line to the point P. The intersection of this line with the elliptic curve produces $Q^{'}$, reflected on the x-axis it yields the point Q. The Figure 2.3 shows the process.

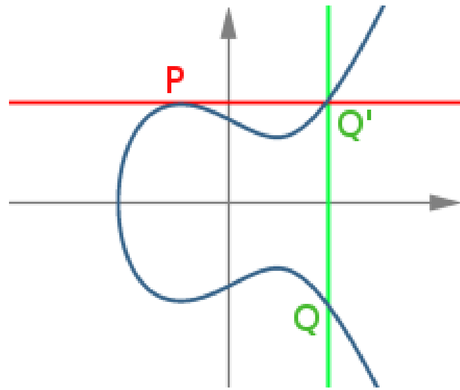


Figure 2.3: Point doubling on a elliptic curve

Definition 2.2.1. *Group law [24]*
The group law defines operations on points which lie on an elliptic curve satisfying the Weierstraß equation $y^2 = x^3 + ax + b$ and have char$(\mathbb{F}_{p^k}) \neq 2, 3$.

- ***Identity*** *∞ is also known as point at infinity. $P + \infty = \infty + P = P, \ \forall P \in E(\mathbb{F}_{p^k})$.*
- ***Negatives*** *$\forall P = (x, y) \in E(\mathbb{F}_{p^k}) \ \exists -P = (x, -y) \in E(\mathbb{F}_{p^k})$. The point $-P$ is called negative of P, note that $(x, y) + (x, -y) = \infty$ and $-\infty = \infty$.*

- ***Point addition*** *Let* $P_1 = (x_1, y_1), P_2 = (x_2, y_2) \in E(\mathbb{F}_{p^k})$*, where* $P_1 \neq \pm P_2$*. Then* $P_1 + P_2 = Q$*, where* $Q = (x_3, y_3) \in E(\mathbb{F}_{p^k})$ *and*

$$x_3 = (\frac{y_2 - y_1}{x_2 - x_1})^2 - x_1 - x_2 \bmod r \textit{ and}$$
$$y_3 = (\frac{y_2 - y_1}{x_2 - x_1})(x_1 - x_3) - y_1 \bmod r.$$

- ***Point doubling*** *Let* $P = (x_1, y_1) \in E(\mathbb{F}_{p^k})$*, where* $P \neq -P$*. Then* $2 \cdot P = (x_2, y_2) \in E(\mathbb{F}_{p^k})$*, where*

$$x_2 = (\frac{3x_1^2 + a}{2y_1})^2 - 2x_1 \bmod r \textit{ and}$$
$$y_2 = (\frac{3x_1^2 + a}{2y_1})(x_1 - x_2)y_1 \bmod r.$$

When a point should be multiplied by an integer $Q = n \cdot P$ the program performs n additions:

$$Q = \underbrace{P + P + \ldots + P}_{n \textit{ times}}.$$

2.3 Zero-knowledge Protocols

Zero-knowledge protocols are an important role in this work. There are several non-interactive zero-knowledge proofs of knowledge in the implemented scheme of this thesis. The signature generation process is a slightly modified non-interactive zero-knowledge protocol from [33].
Roughly speaking, the zero-knowledge property states that, in a protocol, one party believes another that it has information about a secret without learning anything about this secret through the protocol run.
A zero-knowledge proof is not only useful in cryptographic applications but also in the field of mathematical proof systems. This thesis will focus on the cryptographic aspects of zero-knowledge proof systems only.
Such a protocol must satisfy three properties:

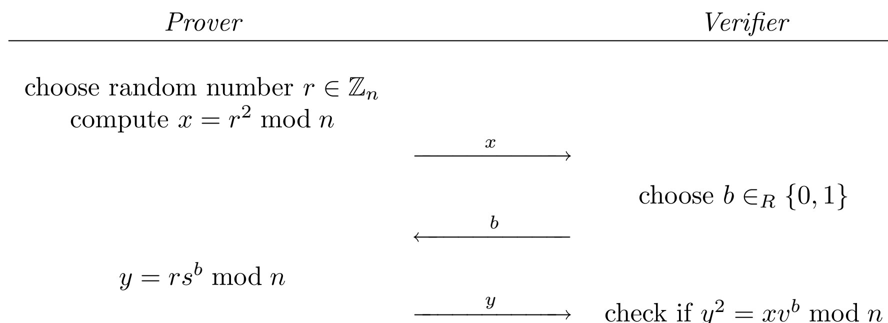
1. **Completeness** If an honest prover knows the secret, he can convince an honest verifier. Honest means that both parties are following the protocol run properly.
2. **Soundness** A cheating prover, without knowing the secret, can convince an honest verifier with a small and negligible probability only.
3. **Zero-knowledge** A cheating verifier cannot learn anything other from the protocol run except that whether the prover knows the secret or not.

To clarify these properties a popular example is given.
One of the most popular zero-knowledge protocols is the Fiat-Shamir algorithm, see [20]. The algorithm is built on the difficulty of computing square roots in $\mathbb{Z}_n^*$ when n is the product of two large primes $n = p \cdot q$. Note that it is impossible to compute a square root in $\mathbb{Z}_n^*$, if the factorization of n is not given.
The Fiat-Shamir algorithm consists of two phases, the key generation phase and the application phase.
In the key generation phase, the prover randomly chooses two large primes p and q and computes $n = pq$. Next, the verifier chooses a secret $s \in \mathbb{Z}_n$ and computes $v = s^2 \bmod n$. Now the prover publishes n and v.
The aim of the application phase is that the prover convinces a verifier that he knows s without revealing s. Therefore the protocol shown in Table 2.1 is executed.

In a security proof, we must check the three properties.
Completeness If the prover knows the secret s, the verifier will be convinced of this fact and the equation is true:

$$y^2 \equiv (rs^b)^2 \equiv r^2 s^{2b} \equiv r^2 v^b \equiv xv^b \bmod n.$$

Soundness A cheating prover can only send the right answers to one of the challenges $b = 0$ or $b = 1$. If the prover could answer both requests (send either



Prover		*Verifier*
choose random number $r \in \mathbb{Z}_n$ compute $x = r^2 \bmod n$		
	$\xrightarrow{x}$	
		choose $b \in_R \{0, 1\}$
	$\xleftarrow{b}$	
$y = rs^b \bmod n$		
	$\xrightarrow{y}$	check if $y^2 = xv^b \bmod n$

Table 2.1: Example of a Zero-knowledge Protocol

y_0 or y_1), he would already know a square root of v: through $y_0^2 = x$ and $y_1^2 = xv$ the next equation follows: $(\frac{y_1}{y_0})^2 = v$. Therefore $(\frac{y_1}{y_0})$ is already a square root of v modulo n. So the probability of a cheating prover is at most $\frac{1}{2}$.
On the other hand, the probability for the prover to cheat is at least $\frac{1}{2}$. If the prover guesses at the beginning of the protocol which b the verifier will send, he can prepare the x upfront. The prover computes $x = r^2 v^{-b'} \bmod n$ and sets $y = r$. If the verifier sends $b = b'$, the prover can answer with $y = r$ and the verifier will not notice the difference.
This shows that the cheating probability for the prover is $= \frac{1}{2}$.
In the security proof the verifier is substituted by a so-called knowledge extractor, that can rewind the prover. The knowledge extractor stores y_0 where $b = 0$ and then rewinds the prover and sends $b = 1$, now storing y_1. Note that now the extractor can compute the secret. So if the knowledge extractor can extract the secret, the prover must posses the secret and therefore the soundness property is proved.
After t protocol runs, the probability is $(\frac{1}{2})^t$ that the prover is cheating, which is negligible in practice.
Zero-knowledge To prove this property the prover is substituted by a so-called simulator which does not know the secret s. If the recorded protocol run is not distinguishable from a real protocol run, the zero-knowledge property is fulfilled. The simulator first chooses a bit $c \in_R \{0, 1\}$ and $r \in_R \mathbb{Z}_n$ and computes $x = r^2 v^c \bmod n$. Now it sends x to the verifier and gets b back.
If $b = c$ the simulator sends $y = r$ and the protocol run will be recorded, else the whole simulator and the verifier are reset and started again.
The so recorded simulated protocol runs contain the same random distribution of (x, b, y) as the recorded normal protocol runs. Every tuple fulfills the equation $y^2 = xv^b$, so the simulated protocol runs are indistinguishable from the normal protocol runs. No one gets any information when observing the protocol or acting as a verifier.

2.3.1 Non-interactive Zero-knowledge Protocols

This work focuses on non-interactive zero-knowledge proofs to prove knowledge of certain values. To transform a zero-knowledge proof into a non-interactive protocol, the bit b is not chosen randomly but computed by a hash function. The global parameters are n and $v = s^2 \bmod n$. The protocol is now defined as follows:

Prover	$n, v = s^2 \bmod n$	*Verifier*
choose random number $r \in \mathbb{Z}_n$		
compute $x = r^2 \bmod n$		
compute $b = \mathcal{H}(x \mid n \mid v)$		
$y = rs^b \bmod n$		
	$\xrightarrow{x,y}$	compute $b^{'} = \mathcal{H}(x \mid n \mid v)$
		check if $y^2 = xv^{b^{'}} \bmod n$

Table 2.2: Example of a non-interactive Zero-knowledge Protocol

Due to the combination of x and n in the hash function, it is assured that the prover cannot choose b arbitrarily. Security of this protocol now holds if we assume the output of $\mathcal{H}$ to be truly random. We then prove security in a so-called random oracle model.
To transform this non-interactive Zero-knowledge protocol into a signature algorithm, the message m, which should be signed, must be included in the hash function $b = \mathcal{H}(x \mid n \mid v \mid m)$.

2.4 Complexity Assumptions

Closely related to the discrete logarithm problem (DLP) is the Computational Diffie-Hellman problem (CDH). Both state as follows.

Definition 2.4.1. *Discrete Logarithm Problem (DLP)*
Given a prime p, a generator g of $\mathbb{Z}_p^*$, and an element $a \in \mathbb{Z}_p^*$, find the integer x, $0 \leq x \leq (p-2)$, such that $g^x = a \bmod p$.

The Discrete Logarithm assumption in $\mathbb{G}_1$ is as follows.

Definition 2.4.2. *Discrete Logarithm assumption in $\mathcal{G}_1$ (DL$\mathbb{G}_1$)*
Let G output descriptions of a bilinear group as defined in Definition 2.1.6 and let Q be a random generator of $\mathbb{G}_1$. For every PPT algorithm $\mathcal{A}$, the following function $Adv_{\mathcal{A}}^{DL}(l)$ is negligible.

$$Adv_{\mathcal{A}}^{DL}(l) = Pr[\mathcal{A}(t, Q, xQ) = x]$$
$$\text{where } t = (p, \mathbb{G}_1, \mathbb{G}_T, e, P) \leftarrow \mathcal{G}(1^l), Q \leftarrow \mathbb{G}_1 \text{ and } x \leftarrow \mathbb{Z}_p^*.$$

Definition 2.4.3. *Computational Diffie-Hellman problem (CDH)*
Given a prime p, a generator g of $\mathbb{Z}_p^*$ and g^x, g^y where $x, y \in \mathbb{Z}_p^*$, compute g^{xy}.

The Decisional Diffie-Hellman (DDH) assumption holds in multiplicative groups. It may also be true for many other cyclic groups of prime order, such as the subgroup of order p of group $\mathbb{Z}_{p'}$, where $p, p^{'}$ are large primes and $p \mid p^{'} - 1$.

Definition 2.4.4. *Decisional Diffie-Hellman (DDH)*
Let p be a prime and $\mathbb{G} = (\mathbb{Z}/p\mathbb{Z})$ a cyclic group with order p. Let g be a generator of $\mathbb{G}$, so that every element in $\mathbb{G}$ can be written as $a = g^b \bmod p$. Given three elements $x = g^c \bmod p$, $y = g^d \bmod p$ and $z \in \mathbb{Z}_p^*$, where c and d are unknown, decide whether $z = g^{cd} \bmod p$ or not.
The DDH problem in bilinear groups is defined as follows. Again group $\mathbb{G}_1$ is an elliptic curve, $\mathbb{G}_T$ is a multiplicative group and e defines a bilinear map $e : \mathbb{G}_1 \times \mathbb{G}_1 \rightarrow \mathbb{G}_T$.
For every PPT algorithm $\mathcal{A}$, the following function $Adv_{\mathcal{A}}^{DDH}(l)$ is negligible.

$$Adv_{\mathcal{A}}^{DDH}(l) = \mid Pr[\mathcal{A}(t, \Phi, \Phi^r, \Phi^x, \Phi^{rx}) = 1] - Pr[\mathcal{A}(t, \Phi, \Phi^r, \Phi^x, \Phi^s) = 1] \mid$$
$$\text{where } t = (p, \mathbb{G}_1, \mathbb{G}_T, e, P) \leftarrow \mathcal{G}(1^l), \Phi \leftarrow \mathbb{G}_T^* \text{ and } x, r, s \leftarrow \mathbb{Z}_p^*.$$

The q-SDH definition was introduced by Boneh and Boyen in 2004 [10]. It states that there is no PPT algorithm that can compute a pair $(c, \frac{1}{s+c}P)$ from a tuple $(P, sP, \ldots, s^q P)$, where $c \in \mathbb{Z}_r$ and $s \in_R \mathbb{Z}_p$.

Definition 2.4.5. *q-Strong Diffie-Hellman (q-SDH)*
For every PPT algorithm $\mathcal{A}$, the following function $Adv_{\mathcal{A}}^{q-SDH}(l)$ is negligible.

$$Adv_{\mathcal{A}}^{q-SDH}(l) = Pr[(\mathcal{A}(t, sP, \ldots, s^qP) = (c, \tfrac{1}{s+c}P)) \wedge (c \in \mathbb{Z}_p)]$$
$$where\ t = (p, \mathbb{G}_1, \mathbb{G}_T, e, P) \leftarrow \mathcal{G}(1^l)\ and\ s \leftarrow \mathbb{Z}_p.$$

In 2004, Boneh and Boyen [9] also presented the DBDH assumption, which intuitively states that there exists no PPT algorithm that can distinguish between a tuple $(aP, bP, cP, e(P,P)^{abc})$ and a tuple (aP, bP, cP, Φ), where $\Phi \in_R \mathbb{G}_T^*$ and $a, b, c \in_R \mathbb{Z}_p^*$.

Definition 2.4.6. *Decisional Bilinear Diffie-Hellman (DBDH)*
For every PPT algorithm $\mathcal{A}$, the following function $Adv_{\mathcal{A}}^{DBDH}(l)$ is negligible.

$$\begin{aligned} Adv_{\mathcal{A}}^{DBDH}(l) =\mid & Pr[\mathcal{A}(t, aP, bP, cP, e(P,P)^{abc}) = 1] \\ & -Pr[\mathcal{A}(t, aP, bP, cP, \Phi) = 1] \mid \end{aligned}$$
$$where\ t = (p, \mathbb{G}_1, \mathbb{G}_T, e, P) \leftarrow \mathcal{G}(1^l), \Phi \leftarrow \mathbb{G}_T^*\ and\ a, b, c \leftarrow \mathbb{Z}_p^*.$$

Nguyen presented in [33] a decisional Diffie-Hellman variant assumption and showed that it is weaker than a DBDH assumption. The following assumption is very similar to the DDH assumption, but elements of multiplicative and additive groups are included.

Definition 2.4.7. *Decisional Diffie-Hellman Variant (DDHV)*
For every PPT algorithm $\mathcal{A}$, the following function $Adv_{\mathcal{A}}^{DDHV}(l)$ is negligible.

$$\begin{aligned} Adv_{\mathcal{A}}^{DDHV}(l) =\mid & Pr[\mathcal{A}(t, P, rP, e(P,P)^x, e(P,P)^{xr}) = 1] \\ & -Pr[\mathcal{A}(t, P, rP, e(P,P)^x, e(P,P)^s] = 1] \mid \end{aligned}$$
$$where\ t = (p, \mathbb{G}_1, \mathbb{G}_T, e, P) \leftarrow \mathcal{G}(1^l)\ and\ x, r, s \leftarrow \mathbb{Z}_p^*.$$

3 Authentication

Authentication is the process of verifying the identity of a person that is trying to get access on some resources. Authentication identifies an individual and refers to whether the source and contents of a message are what they claim to be.
There are different kinds of authenticators. They can be divided into three subgroups.

- The individual can be linked to confidential information that only he or she is supposed to know, for example a password, private key or PIN.
- The individual can be associated with a unique logical device or logical address, like a MAC address or a IMSI number on the SIM card of a mobile phone.
- A unique attribute of the individual like the voice or fingerprint is used for authentication.

In most private and public networks (including the Internet) authentication is commonly performed by the use of passwords. The weakness of this authentication method is that passwords can be stolen, revealed, or forgotten. Due to many different malicious threats, like viruses and phishing scams, the potential to loose a password to an adversary increases.
To encounter this problem, other, more stringent methods are required. The identification by the use of a unique logical device is problematic. First, we must use a device everyone has got. Second, the unique information must be transported secretly and without loss of integrity to the target authentication service. There must also be a way to secure that the unique information cannot be changed by an adversary.
A better way to authenticate an individual is the use of biometrics. The attributes, like a palm print, cannot easily be imitated, but there must be a physical device to gather this information from the user. Even if the physical device is present, the attribute must be securely transported over a network.
If no devices are presented to gather biometric information or to provide a unique identification, software can be used to accomplish security. The authenticity of a message and the authentication of a user can be achieved by using digital signatures and certificates. Although these procedures are not completely secure, they provide flexibility, platform independence and a high security standards.
The cornerstone of most cryptosystems is authentication. Only if users can authenticate themselves against a server, it will be able to provide trusted users

the correct values. Without authentication, other requirements like integrity or confidentiality are no longer relevant.

Anonymity is another requirement in cryptographic systems. It comes into play when the identity of individuals in a certain event should be kept secret. In a wide range of applications this property is very desirable or needed, like voting or anonymous donations.
The anonymity property critically depends on the position of the adversary. For example, in case of an anonymous bulletin board, a posting by a member is kept anonymous to the other members of the board. But it is possible that the administrator of the board has some privileged information to identify the member who posted the message.
Often, in practice, a user gets a unique identifier, called a pseudonym. With this pseudonym a long-term relationship can be established with an entity. The user is anonymous in respect of its identity to other users as long as they cannot link the pseudonym to an identity.
Instead of giving each user full anonymity or pseudonymity, there is the concept of anonymity with respect to a group. The identity of the user cannot be revealed by anyone except a small group of trusted administrators, but everyone can identify the group to which the user belongs to.
Almost every application which provide the anonymity property also needs authorization. This ensures that only a certain group of users can use the application and that the provider can seek a user if the anonymity was misused. Most schemes only provide either anonymity or authentication. In this thesis we implement a scheme that provide both security properties at the same time.

Most systems, which provide signer-anonymity, usually do not address mechanisms that allow a dedicated third party to revoke the anonymity in cases of a dispute. If the process of revoking should be successful, first of all the member who sent the message must be traced. To establish traceability, the unique identifier of a member is encrypted and hidden in the signature. Therefore only an administrator with privileged information can decrypt this piece of information and can link an authorized identity to the anonymous signature.

3.1 Digital Signature Schemes

A digital signature is the digital analogue of a handwritten signature. Informally, a digital signature is a bit string that connects a message to a public key of the signer, so the receiver can believe that the message was really sent by the signer. As long as the private key of the signer remains secret, the digital signature provides non-repudiation, meaning that the signer cannot successfully claim that he did not sign a message, when it can be verified with his public key.
There are ambitions to use digital signatures in the public sector to replace the handwritten signature and build up an online government agency. The potential use of digital signatures is not exhausted yet.
The whole concept of digital signatures was put forth by Diffie and Hellman their seminal paper [18], in 1976.

Definition 3.1.1. *A digital signature scheme*
A digital signature scheme is a triple of algorithms SIG = (KeyGen, Sign, Verify). The first and the second one are probabilistic, while the third one is deterministic. The algorithm KeyGen generates a secret key x_s and a corresponding public key y_s of a signer S on input of the system parameters. The algorithm Sign takes x_s and a message m as input and outputs a signature σ of m. On input of a message m, a signature σ, and the public key y_s of a signer, the algorithm Verify outputs true or false. The following must be satisfied.

$$Verify(m, \sigma, y_s) = \begin{cases} true & if\ Prob\ \sigma = Sign(m, x_s) > 0 \\ false & otherwise \end{cases}$$

Furthermore, a signature scheme must be unforgeable. This means that it must be infeasible to compute a signature of a message with respect to a public key without knowing the corresponding secret key.

The following standard definition is needed in security proofs of signature schemes.

Definition 3.1.2. *Unforgeability against Chosen Message Attacks [11]*
A signature scheme SIG = (KeyGen, Sign, Verify) is (t, q, ϵ)-existentially unforgeable against adaptive chosen message attacks, if any adversary with runtime t wins in the following game with probability at most ϵ after issuing at most q signing queries.

1. ***Setup***. *The challenger first runs KeyGen that outputs a secret and a public key. The public key is given to the adversary.*
2. ***Signature Queries***. *The signature queries* $m_0, \ldots, m_q$ *are sent from the adversary to the challenger. The challenger computes for each query* m_i *a signature* σ_i *by calling the Sign functionality. Then each signature* σ_i *is sent to the adversary. These queries may be asked adaptively so that each query* m_i *may depend on the replies of the queries* $m_0, \ldots, m_{i-1}$.
3. ***Output***. *Finally, the adversary computes a pair* (m, σ) *and publishes it. The adversary wins if* σ *is a valid signature of the message* m *according to the functionality Verify and* (m, σ) *is not among the pairs* (m_i, σ_i) *generated during the query phase.*

3.2 Digital Certificates

In a digital certificate a trusted third party binds a user's identity to his public key. Technically, the trusted third party just signs the user's public key together with some identifying information. Digital certificates are useful for large scale public key infrastructures (PKIs). They are one of the most common ways to securely distribute public keys over a large and untrustworthy network. With these public keys, it is possible to verify a digital signature.
So if Bob wants to send secret information to Alice, he can just download Alice's public key and encrypt the message. But in a large network Charlie could also publish his key and claim that this key is the original key of Alice. If Bob believes Charlie and uses Charlie's public key, Charlie could intercept the ciphertext, decrypt it and read the secret information. So Bob must be sure that he gets the correct key. A way to accomplish this is to ask a trusted third party, a certificate authority (CA). This CA will send Bob

- a public key,
- a name of the person or cooperation, belonging to the public key,
- an information about how long the certificate will be valid,
- a referrer (URL) to the location of a revoke list,
- and a digital signature of the all bullets above, signed with the CA's private key.

An example certificate is given in chapter 6.2.1.
If the CA is trusted, everyone will be sure that he is getting the correct public key of a given entity. The CA's public keys must be stored on Bob's computer, so that he can verify the certificate. Usually, public keys of CAs are distributed by embedding them into web browsers. The resulting trust hierarchy is called

public key infrastructure (PKI) and is described in Section 3.3.

Digital certificates are a way to transfer public keys over an untrusted network. With these keys a digital signature can be verified or a message can be encrypted. An asymmetric encryption algorithm is used to accomplish this. To proof the security of an asymmetric encryption algorithm, first a general encryption scheme is defined below.

Definition 3.2.1. *Encryption Scheme*
A triple (KeyGen, Enc, Dec) is an encryption scheme, if KeyGen and Enc are PPT algorithms and Dec is a deterministic polynomial-time algorithm which satisfies the following conditions:

- *On input 1^n, algorithm KeyGen outputs a pair of bit strings.*
- *For every pair (e, d) in the range of KeyGen(1^n), and for every $m \in \{0,1\}^*$, algorithm Enc (called encryption) and Dec (called decryption) satisfy:*

$$Pr[m = m^{'}; m^{'} \leftarrow Dec_d(Enc_e(m))] = 1.$$

The security of an asymmetric algorithm can be described in the following definition.

Definition 3.2.2. *Indistinguishability under chosen-plaintext attack (IND-CPA)*

For an asymmetric key encryption algorithm, this definition is formalized by a game between an adversary and a challenger.
The adversary is a probabilistic polynomial time Turing machine, so the computation must be complete in a polynomial number of time steps. In the following game, $E_{pk}(m)$ represents the encryption of m under the public key pk*.*

1. *The challenger generates a public key* pk *and a secret key* sk*. The input of the generation function is a security parameter* l*. The public key is sent to the adversary.*
2. *The adversary can make any number of encryption with* pk *or execute any number of other operations.*
3. *Now the adversary sends two distinct plaintexts m_0, m_1 to the challenger.*
4. *A bit $b \in_R \{0,1\}$ is chosen randomly by the challenger and the challenge $c = E_{pk}(m_b)$ is sent to the adversary.*
5. *The adversary can perform any number of computations or encryptions. Finally it outputs a guess for b.*

An encryption algorithm is indistinguishable under chosen-plaintext attack if a probabilistic polynomial time adversary has only a negligible advantage over guessing b randomly. A negligible advantage is a win probability for the adversary of $\frac{1}{2} + \epsilon(l)$*, where* $\exists\ l_0 : \epsilon(l) < \frac{1}{poly(l)}\ \forall\ l > l_0$*.*
Although the adversary can compute $c_0 = E_{pk}(m_0)$ *and* $c_1 = E_{pk}(m_1)$ *and can compare both with c, due to the probabilistic nature of E, this does not increase the advantage of the adversary.*

3.3 Public Key Infrastructures

A digital certificate is a digital signature, issued by an entity or authority, stating that a public key belongs to a specific entity. So Alice can trust a certificate issued by an authority X for user Bob if and only if the following conditions are met:

- Alice possess the public key of X and knows that it is authentic.
- Alice trusts X to be honest and to sign only authenticated public keys.

In case Alice does not possess an authentic copy of the public key of X, the first condition can be satisfied by using a certificate of X's public key issued by another authority Y. This process can be iterated, generating a chain of certificates. These chains of certificates establish a public key infrastructure.
A PKI usually combines the functions issue, revoke, store and retrieve of digital certificate. There are four entities in the infrastructure:

- Certification Authority: root of the infrastructure
- Registration Authority: an optional system for certain management functions
- End Entity: end user systems that use a certificate
- Database: a system that stores certificates and certificate revoke lists (CRLs)

3.3.1 Drawbacks of PKIs

A big drawback of public key infrastructures and digital certificates is the revocation mechanism of members. It is very easy to add a new member to a group. The new user just gets a new certificate. A group is, in this case, defined as all certificates which are signed by a so called root certificate. Root certificates are owned by a trusted third party like an authentication service or a CA. A very popular one is VeriSign [38].
To remove a member, his certificate is stored in the CRL (Certificate Revoke List). So basically, each time a user is connecting to a web site and receiving a

digital certificate, he must also connect to the authentication service and download the newest CRL to be sure that the certificate of the web site has not been revoked.
Few software implementations for digital certificates provide this functionality. A drawback of CRLs is the dependency on the number of users revoked, because each revoked certificate creates another entry in this list. Current systems use a workaround solution, the restriction of the validation time. The maximum validation time of certificates is usually one to three years from the day of purchase. If the authentication service wants to revoke a user, no more certificates will be sold to him when his certificates have expired.

3.4 Group Signatures Schemes

Today, it is easy to provide confidential communication in large networks (e.g. using the TLS/SSL protocol suite that is implemented in standard browsers), but without signer-anonymity an adversary may construct a web profile for users. For example, if a user makes a medical statement in an online forum and claims he is a doctor, using a digital signature, everyone could discover who is behind the statement. Only a scheme with sender-anonymity can be used to protect the identity of a person. And the traceability requirement can deter a user from abusing the given anonymity. These security requirements can concurrently be accomplished with a group signature scheme.
A group signature scheme is a special signature scheme. A member of a group can anonymously sign a message on behalf of the group. This idea was first published by Chaum and van Heyst [15] in 1991.
A group signature, just as any other digital signature scheme, lets the signer demonstrate some secret knowledge in respect of a certain document. It is possible for everyone to verify a given group signature. Unlike other signature schemes no one, with the exception of a dedicated group administrator, can reveal the identity of the signer. Furthermore it is impossible to decide whether two signatures originate from the same user. The corresponding property is called unlinkability and can be understood as a very strong notion of sender-anonymity. While the group administrator is able to open a signature and identify the signer, it is not possible for anyone, including the group administrator, to impersonate another group member.
Formally, a group signature scheme must have the following three properties:

1. only members of the group can sign messages;

2. the receiver of the signature can verify that it is a valid signature of that group, but cannot discover which member of the group made it;

3. in case of dispute the group manager can "open" the signature to reveal the identity of the signer.

This makes group signatures an interesting building block for e-voting applications.
The group manager or administrator is essential for the group signature scheme because he is the only one who can add or delete members to/from the group and revoke the anonymity of members. So basically, everyone in the group must trust the group administrators.

3.4.1 Formal Definition of Group Signatures

Jan Camenisch [13], [2] defined a generalized concept of group signatures. Typically these schemes are defined as follows.

Definition 3.4.1. *Group signatures*
A group signature scheme is a digital signature scheme comprised of the following procedures:

- ***Setup**: A probabilistic algorithm which outputs the initial public group key γ and the secret group administrator keys ik and ok after the input of the security parameter l.*
- ***Join**: An interactive protocol between the group administrator, who takes the secret key ik as input, and a user that results in the user becoming a new group member. The new member gets a membership secret x_i and a membership certificate.*
- ***Sign**: A probabilistic algorithm that takes as input the group public key γ, a membership certificate, a membership secret x_i, and a message m and outputs a group signature s of m.*
- ***Verify**: An algorithm which tests an alleged group signature s of a message m with respect to a group public key γ.*
- ***Open**: An algorithm which outputs the identity of the signer and a proof of this claim after the input of a message m, a valid signature s on this message, a group public key γ, and the secret group administrator key ok.*

A group signature scheme must satisfy the following properties:

- *Correctness: Signatures produced by a valid group member with **Sign** must be accepted by **Verify**.*
- *Anonymity: Given a valid signature s of some message m, identifying the original signer is computationally hard for everyone but the group administrator, the "opener".*
- *Traceability: The group administrator is always able to open a valid signature and identify the actual signer. Therefor, any colluding subsets of group members cannot generate a valid signature that the "opener" cannot link to one specific group member of that subset.*

- *Non-frameability: No one can produce a valid signature s to a message m, which opens to a group member, who did not sign m.*

Most group signature schemes, like [4], do not need a revoke algorithm because they do not explicitly support dynamic groups. This means that the group size and therefor the maximum number of members is static and cannot be changed later. All membership certificates are generated in the **Setup** algorithm and are only given to the user in the **Join** protocol.
In a fully dynamic group the number of members who can join the group is not restricted. There is also a way to revoke members from the group. This is an important when group members misuse the anonymity guaranteed by the system, for example by signing illegal contents. So in a fully dynamic group, as assumed in this thesis, we need an additional algorithm **Revoke**.

Definition 3.4.2. *Revoke algorithm*
This algorithm removes a member from the group.

- ***Revoke**: An algorithm which removes a member from the group. The input is a group public key γ and the secret group administrator key ik. The output is a new group public key.*

Another serious problem is a dishonest group administrator. The first group signature schemes, like [15], only have one group administrator, who has got the opening key *ok* and the issuing key *ik*. So one entity alone can issue new members and can revoke the anonymity property. A solution to this problem is the splitting of these administrative functions onto different management entities in the group. Should the opener be dishonest, he could falsely accuse an arbitrary user to have signed a specific message or could claim that a signature cannot be opened to protected a user. A solution to the dishonest opener problem is the **Judge** algorithm. The opener must produce a proof of his claim which can be verified by any user using the **Judge** procedure.

Definition 3.4.3. *Judge algorithm*
This algorithm is used to verify a proof produced by an opener.

- ***Judge**: A probabilistic algorithm that on input a group public key γ and a proof from the **Open** algorithm verifies this proof.*

If the issuer is dishonest, he could use the membership secret x_i and the membership certificate to impersonate this member. To prevent this attack, only the member must know the value x_i. A possible solution is the use of an according zero-knowledge protocol, such that the group administrator is convinced that the member has chosen a valid x_i but does not know the exact value.
Additionally, "long-term credentials", as stated by [2], are necessary to protect group members from being framed by a corrupt issuer. This can be established by using an independent PKI and certificates for every member. Consequently each member and potential member has a certified public key and a matching private key that is kept secret. In combination a PKI is used for authentication,

and the group signature scheme provides anonymous and traceable authentication at the group level.

3.4.2 Overview of the Presented Group Signature Scheme

Our new scheme which is presented in detail in the next chapters, combines a PKI and a group signature scheme. This combination has advantages over many other schemes.
Apart from the security requirements which the scheme provides, the signature size is approximately as large as the size of a standard signature, for more details see Appendix B.
With this new scheme it is also possible to distribute a large number, e.g. thousands, of keys to members, so that members can sign information in behalf of the group. Distribution and administration of so many users is not always possible in every software implementation of signature algorithms.
In a larger network, it is also possible to use tree structures for the servers. Such an infrastructure can decrease the latency time, and different subgroups could split up to form a new group with an accumulator. For example, if in a corporation every department has its own server for accumulator updates and join requests, a department can be detached and can get its own accumulator value. With such a structure, problems like certificate revoke lists, see drawbacks of PKIs 3.3.1, are not given in this scheme. Also, nearly no operation depends on the number of members or the number of revoked members. This advantage is not given in most other signature schemes. The use of validation times is no longer necessary because there are procedures which provide an easy way to immediately revoke members.
Finally, the presented scheme is provably secure. The main signature protocol and also the underlying zero-knowledge protocols are secure and the related security proofs are given in this document.

3.4.3 Security Requirements

The security requirements are based on models, which were first presented by Bellare, Shi and Zhang [6]. There are four security requirements: correctness, anonymity, traceability and non-frameability.
To prove the presented scheme secure with respect to these requirements, we will present several experiments in which the adversary has access to specific oracles. The specific description of the algorithms mentioned below is given in 5. We first introduce these oracles.

- **AddU($\cdot$)** The add user oracle takes as input an integer $i \in \mathbb{N}$. An adversary can add i to the honest user group. The oracle generates a new PKI certificate and executes the **Join** protocol, on behalf of i and the issuer.

This oracle simulates the entire **User PKI certificate generation** and the **Join** protocol. If the issuer accepts the protocol run, i will be added to the database. If the prospective member also accepts, its final state will be recorded as the group secret key $gsk[i]$ and the witness $w[i]$. The adversary only knows the public key of the PKI certificate, but not the transcript generated by the oracle.

- **CrptU(·,·)** By calling the corrupt user oracle with the arguments $i \in \mathbb{N}$ and a public key of a PKI certificate chosen by the adversary, the adversary can corrupt a user i. Note that the user i is not a member of the group yet. The oracle initializes the issuer to be ready to participate in a **Join** protocol with i.

- **SndToI(·,·)** Having corrupted user i, the adversary can now use the send to issuer oracle to engage in the **Join** protocol with an honest issuer. The adversary must not necessarily execute the described **Join** protocol, it can provide the oracle with i and a message M_{in} which will be sent to the issuer from the user i, instead of the normal **Join** protocol. The issuer is managed by the oracle, which computes a response using the **Join** protocol and returns the outgoing message to the adversary. At last, the oracle updates the database if the issuer accepted the protocol run.

- **SndToU(·,·)** There are some cases in which the issuer could be corrupted. To simulate this situation, the send to user oracle can be used by the adversary, to engage in the **Join** protocol with an honest user as a corrupted issuer. The adversary gives to the oracle i and M_{in}, the message to be sent to user i. The oracle manages the honest user and computes a response using the **Join** protocol. After the simulated protocol run, the oracle will return the outgoing messages to the adversary and will set the user's group secret key $gsk[i]$ and witness $w[i]$, if the user accepted the protocol run.

- **USK(·)** The adversary can call the user secret keys oracle with $i \in \mathbb{N}$ to get both the user's group secret key $gsk[i]$ and the user's private key of the PKI certificate.

- **RReg(·)** The adversary can read the contents of the registration database at position i with the read registration oracle.

- **WReg(·)** There are some definitions in which the adversary can alter the registration database at position i by calling the write registration oracle.

- **GSig(·,·)** The signing oracle can be used to make a user i sign a message m and send it to the adversary, as long as i is an honest user.

- **Ch(b,·,·,·)** A challenge oracle allows an adversary to attack anonymity. It depends on a challenge bit b set by the overlying experiment. The adversary provides a pair of identities i_0, i_1 and a message m, and obtains a signature of m under the private key of i_b, as long as the users i_0, i_1 are honest users.

- **RevokeU($\cdot$)** The revoke user oracle lets the adversary remove user i from the group.
- **Witness($\cdot$)** The witness oracle returns the witness of a user i.

The following security requirements are modeled with the defined oracles.

- *Correctness* This requirement ensures that the signature scheme will output the correct values if all parties in the scheme are honest. The following conditions must be fulfilled: the signature should be valid; the **Open** algorithm, given the correct message and signature, should identify the signer; and the **Judge** algorithm should accept the proof returned from the **Open** algorithm. These terms must hold for all honest users and for any variant they were added or deleted to/from the group.
  The adversary has access to the **AddU($\cdot$)** and **RReg($\cdot$)** oracles. The adversary creates an honest group member who signs a message. The correctness condition holds if none of the following steps fail: an honest verifier accepts the signature; the **Open** algorithm returns a correct group member; and the **Judge** algorithm accepts the proof returned by the **Open** algorithm. Note that the adversary is computationally not bounded.
- *Anonymity* In this experiment the adversary is polynomial time (PT) restricted. The adversary knows the issuing key (s, k) and has access to the **Ch(b,$\cdot$,$\cdot$,$\cdot$)**, **SndToI($\cdot$,$\cdot$)**, **SndToU($\cdot$,$\cdot$)**, **CrptU($\cdot$,$\cdot$)**, **WReg($\cdot$)**, **RevokeU($\cdot$)**, **USK($\cdot$)** and **Witness($\cdot$)** oracles. Formally, the anonymity condition holds if the probability that the adversary can correctly guess the bit b after an oracle call **Ch(b,$\cdot$,$\cdot$,$\cdot$)** is negligible. The adversary does not have to recover the identity of the signer of a signature, but needs to distinguish which one of two signers signed a target message of the adversary's choice.
  The adversary is provided with extremely strong attack capabilities, but notice that the adversary must not send queried identities i_0, i_1 to the **RevokeU($\cdot$)** oracle, and that the opener is uncorrupted.
- *Traceability* In this experiment the adversary must be unable to produce a signature which an honest opener cannot link to a member of the group, or an honest opener believes to have found the right member of the group but cannot produce a proof for this claim. In this experiment, the adversary has access to the **AddU($\cdot$)**, **RReg($\cdot$)**, **SndToI($\cdot$,$\cdot$)**, **USK($\cdot$)**, **CrptU($\cdot$,$\cdot$)**, **RevokeU($\cdot$)** and **Witness($\cdot$)** oracles and knows the opening key $x^{'}$. The adversary is PT restricted.
  Some capabilities are denied to the adversary. It cannot corrupt the issuer and the opener can only be partially corrupted, this means that the adversary knows $x^{'}$ but the opener operates correctly.
- *Non-frameability* The adversary tries to output a message, a signature, an identity and an opening proof. The adversary wins if the signature is

valid for the specific message, the identity is the identity of an honest user i and the judge accepts the opening proof and believes that the honest user i has signed the message.
The adversary can call the **SndToU**$(\cdot,\cdot)$, **WReg**$(\cdot)$, **USK**$(\cdot)$, **CrptU**$(\cdot,\cdot)$, **RevokeU**$(\cdot)$ and **Witness**$(\cdot)$ oracles and knows the opening key $x^{'}$ and the issuing key (s, k). The adversary is PT restricted.
Notice that the adversary cannot use **USK**$(\cdot)$ to obtain the signing key of the user i.

The security proof of our scheme is given in 5.12.

4 Dynamic Cryptographic Accumulators

In this chapter we introduce dynamic cryptographic accumulators. Thereby we closely relate to the work of Fazio and Nicolosi [19].
The concept of accumulators was first introduced by Benaloh and de Mare [7] in 1993. It was proposed as a decentralized alternative to public key infrastructures in the design of secure distributed protocols. In this paper the basic functionalities and security properties of dynamic cryptographic accumulators are described.
Basically, an accumulator combines a large set of values into one short accumulator value. Each member of a group can be mapped to one value in the set of the accumulator group and it is possible to use a short witness value to prove that a specific member is indeed included in the accumulator group.
In 2002 Camenisch and Lysyanskaya [14] introduced the more demanding notion of dynamic accumulators, which enables the dynamic deletion and addition of members from/to the group. This proposed scheme achieves a much higher degree of flexibility, because the deletion and addition functionalities are independent of the number of values in the accumulator and every member is able to update old witnesses. This update can be accomplished without knowing sensitive information, like group administrator keys.
In the same paper [14], Camenisch and Lysyanskaya showed that dynamic accumulators can be used in group signature schemes to apply membership revocation. Also, ID escrow schemes and anonymous credential systems can be expanded with dynamic accumulators to provide membership revocation.
A group signature scheme, see Section 3.4 for more details, allows a member of a group to prove his membership without revealing his identity. To discourage abuse, the group administrator can still discover the identity. As see in Section 5.6.3, the protocol reveals no information about the identity of the group member to anyone except a group administrator, the opener, who can open the protocol transcript with some trapdoor information.
Was there no way to uncover the identity of a member, clearly nothing would prevent a user from misusing the anonymity. So this scheme can only be successful if there is an efficient way to identify and revoke a given member. Therefore the member must be traceable and removable from the group. It must be ensured that these actions are only available for group administrators. Using a dynamic accumulator scheme, revoking a member is as simple as deleting a value from the accumulator.
To prevent users from gathering together to gain extra information, the group

manager must ensure that there is some randomness in the user's secret key x_i. This must be done in a secure way to guarantee that the group manager learns nothing about the user's secret key. This assures that the group manager cannot misuse x_i to impersonate himself as the user at another authority. All this properties must be fulfilled by the join protocol.
A user who has obtained membership in a group can prove this membership via a zero-knowledge proof of knowledge of the hashed message, a part of his secret key and a witness for the current accumulator. For traceability, the proof also includes an encryption of a part of the user's secret key, so that a group administrator, the so called opener of the group, can decrypt this part and identify the member. This is accomplished by linking a decrypted value to a digital certificate, which is created and monitored by a PKI.
In a dynamic accumulator scheme, as presented by Camenisch and Lysyanskaya [14] as an enhancement of [2], the user must also prove that his membership certificate is valid. Therefore the proof of knowledge is extended, so that the user shows that he knows a witness for his identification value and combines this information with the current accumulator. If a group administrator deletes this user from the group and updates the accumulator accordingly, the verifier would correctly reject the proof of the user. For this propose the verifier must make sure that he is using the correct current value of the accumulator.
As described above the dynamic cryptographic accumulator is a very efficient way to construct a group signature scheme and to allow revocation in an anonymous fashion. Revocation becomes a very efficient operation by only deleting one value in the accumulator.

4.1 Definitions of Cryptographic Accumulators

A general definition of accumulators was given by Camenisch and Lysyanskaya [14]. This definition of a secure accumulator is given below.

Definition 4.1.1. *Secure Accumulator*
A secure accumulator for a family of inputs $\{\mathcal{X}_k\}$ is a family of families of functions $\mathcal{G} = \{\mathcal{F}_k\}$ with the following properties:

- *Efficient generation: There is an efficient probabilistic algorithm G that on input 1^k produces a random element f of $\mathcal{F}_k$. Moreover, along with f, G also outputs some auxiliary information about f, denoted aux_f.*
- *Efficient evaluation: $f \in \mathcal{F}_k$ is a polynomial-size circuit that, on input $(u, x) \in \mathcal{U}_f \times \mathcal{X}_k$, outputs a value $v \in \mathcal{U}_f$, where $\mathcal{U}_f$ is an efficiently-samplable input domain for the function f; and $\mathcal{X}_k$ is the intended input domain whose elements are to be accumulated.*
- *Quasi-commutative: For all k, for all $f \in \mathcal{F}_k$, for all $u \in \mathcal{U}_f$, for all*

$x_1, x_2 \in \mathcal{X}_k$, $f(f(u, x_1), x_2) = f(f(u, x_2), x_1)$. If $X = \{x_1, \ldots, x_n\} \subset \mathcal{X}_k$, then by $f(u, X)$ we denote $f(f(\ldots(u, x_1), \ldots), x_n)$.

- *Witnesses: Let $v \in \mathcal{U}_f$ and $x \in \mathcal{X}_k$. A value $w \in \mathcal{U}_f$ is called a witness for x in v under f if $v = f(w, x)$.*
- *Security: Let $\mathcal{U}_f^{'} \times \mathcal{X}_k^{'}$ denote the domains for which the computational procedure for function $f \in \mathcal{F}_k$ is defined (thus $\mathcal{U}_f \subseteq \mathcal{U}_f^{'}, \mathcal{X}_k \subseteq \mathcal{X}_k^{'}$). For all probabilistic polynomial-time adversaries $\mathcal{A}_k$,*

$$Pr[f \leftarrow G(1^k); u \leftarrow \mathcal{U}_f; (x, w, X) \leftarrow \mathcal{A}_k(f, \mathcal{U}_f, u) : \\ X \subset \mathcal{X}_k; w \in \mathcal{U}_f^{'}; x \in \mathcal{X}_k^{'}; x \notin X; f(w, x) = f(u, X)] = neg(k);$$

Note that only the legitimate accumulated values, $(x_1, \ldots, x_n)$, must belong to $\mathcal{X}_k$; the forged value x can belong to a possibly larger set $\mathcal{X}_k^{'}$.

When Benaloh and de Mare [7] first proposed accumulators, they defined them as one-way hash functions.
From the Definition 2.1.7 and the definition for quasi-commutativeness above we can deduce the concept of one-way accumulators.

Definition 4.1.2. *One-Way Accumulators*
A family of one-way accumulators is a family of one-way hash functions, each of which is quasi-commutative.

This definition does not guarantee security in a scenario where an adversary $\mathcal{A}$ actively participates in the group and the values x and w are not random but chosen by the adversary. To encounter this problem, Barić and Pfitzmann [3] proposed the notion of collision-free accumulators:

Definition 4.1.3. *Accumulator Scheme*
*An accumulator scheme is a 4-tuple of polynomial time algorithms **(Gen, Eval, Wit, Ver)**, where:*

- ***Gen**, the key generation algorithm, is a probabilistic algorithm used to set up the parameters of the accumulator. **Gen** takes a security parameter 1^k and an accumulation threshold N (i.e. an upper bound on the total number of values that can be securely accumulated) as input and returns an accumulator key p_k from an appropriate key space $K_{k,N}$;*
- ***Eval**, the evaluation algorithm, is a probabilistic algorithm used to accumulate a set $X \doteq \{x_1, \ldots, x_{N'}\}$ of $N^{'} \leq N$ elements from an efficiently-samplable domain X_{p_k}, where p_k is some accumulator key from $K_{k,N}$. **Eval** receives as input $(p_k, x_1, \ldots, x_{N'})$ and returns an accumulated value (or accumulator) $v \in Z_{p_k}$ and some auxiliary information aux, which will be used by other algorithms. Notice that every execution of **Eval** on the same input $(p_k, x_1, \ldots, x_{N'})$ must yield the same accumulated value v, whereas the auxiliary information aux can differ;*

- ***Wit***, *the witness extraction algorithm, is a probabilistic algorithm that takes as input an accumulator key* $p_k \in K_{k,N}$, *a value* $x_i \in X_{p_k}$ *and the auxiliary information aux previously outputted (along with the accumulator v) by* ***Eval***$(p_k, x_1, \ldots, x_{N'})$, *and returns either a witness* w_i *(from an efficiently-samplable witness space* W_{p_k}*) that proves that* x_i *was accumulated within v if this is indeed the case, or the special symbol* $\perp$ *if* $x_i \notin \{x_1, \ldots, x_{N'}\}$.
- ***Ver*** *the verification algorithm, is a deterministic algorithm that, on input* (p_k, x_i, w_i, v), *returns a Yes/No answer according to whether the witness* w_i *constitutes a valid proof that* x_i *has been accumulated within v or not.*

As a short example the accumulator scheme proposed in [7] is described. A function h_k from the family $\mathcal{H}_\lambda$ of hash functions is randomly selected which suits the appropriate security values. This function along with a $x \in_R X_k$ is the key generation algorithm. To accumulate values $x_1, \ldots, x_N \in X_k$ into the accumulator v the operation $v_i \leftarrow h_k(v_{i-1}, x_i)$ is executed.
To verify a given witness w_i and x_i, the verifier must check if $h_k(w_i, x_i) = v$. This shows that the resulting scheme is correct.

4.2 Definition of Dynamic Cryptographic Accumulators

For many cryptographic applications it is necessary to be able to delete values from the accumulator. The only way to accomplish this task in a static accumulator scheme is to recompute all values. As a solution of this problem, the dynamic accumulator scheme was introduced by Camenisch and Lysyanskaya [14].
A secure accumulator, according to Definition 4.1.1, is called dynamic if the following definition is fulfilled.

Definition 4.2.1. *Secure Dynamic Accumulator*

- *Efficient deletion: there exists efficient algorithms* D, W *such that, if* $v = f(u, X), x, x^{'} \in X$, *and* $f(w, x) = v$, *then* $D(aux_f, v, x^{'}) = v^{'}$ *such that* $v^{'} = f(u, X \backslash \{x^{'}\})$; *and* $W(f, v, v^{'}, x, x^{'}) = w^{'}$ *such that* $f(w^{'}, x) = v^{'}$.

A secure dynamic accumulator can also be defined as a scheme.

Definition 4.2.2. *Dynamic Accumulator Scheme*
A dynamic accumulator scheme is a 7-tuple of polynomial time algorithms ***(Gen, Eval, Wit, Ver, Add, Del, Upd)***, *where:*

- ***Gen, Eval, Wit, Ver*** *are defined in the accumulator scheme above 4.1.3;*
- ***Add****, the element addition algorithm, is a deterministic algorithm that given an accumulator key* p_k*, a value* $v \in Z_{p_k}$ *obtained as the accumulation of some set* X *of less than* N *elements of* X_{p_k}*, and another element* $x^{'} \in X_{p_k}$*, returns a new accumulator* $v^{'}$ *corresponding to the set* $X \in \{x^{'}\}$*, along with a witness* $w^{'} \in W_k$ *for* $x^{'}$ *and some update information* aux_{Add} *which will be used by the* ***Upd*** *algorithm;*
- ***Del****, the element deletion algorithm, is a deterministic algorithm that given an accumulator key* p_k*, a value* $v \in Z_k$ *obtained as the accumulation of some set* X *of elements of* X_{p_k}*, and an element* $x^{'} \in X$*, returns a new accumulator* $v^{'}$ *corresponding to the set* $X \setminus \{x^{'}\}$*, along with some update information* aux_{Del} *which will be used by the* ***Upd*** *algorithm;*
- ***Upd****, the witness update algorithm, is a deterministic algorithm used to update the witness* $w \in W_k$ *for an element* $x \in X_{p_k}$ *previously accumulated within an accumulator* $v \in Z_k$*, after the addition (or deletion) of an element* $x^{'} \in X_{p_k} \setminus \{x\}$ *in* v*.* ***Upd*** *takes as input* (p_k, x, w, b, aux_{op}) *(where op is either* ***Add*** *or* ***Del****), and returns an updated witness* $w^{'}$ *that proves the presence of* x *within the updated accumulator* $v^{'}$*.*

This scheme does not need a trusted authority. The drawback is that everyone is able to delete or add new values from/to the accumulator. To avoid this the accumulator manager executes the **Gen** algorithm and keeps a secret trapdoor information t_k (which is unique for the accumulator key p_k). It can be used to enable the accumulator manager to delete and/or add new values from/to the accumulator.
Camenisch and Lysyanskaya [14] formalized a security definition for a secure dynamic accumulator. It states, that the dynamic accumulator is secure against an adaptive adversary in the following scenario: The group administrator sets up the function f and the value u and the trapdoor information aux_f stays secret. The adversary modifies the table X adaptively. When a value x is added or deleted, the group administrator updates the table X accordingly. Finally the adversary attempts to output a witness that $x^{'} \notin X$ is in the current accumulator v.

Definition 4.2.3. *Let* $\mathcal{G}$ *be a dynamic accumulator algorithm. Let* M *be an interactive Turing machine set up as follows: It receives input* (f, aux_f, u)*, where* $f \in \mathcal{F}_k$ *and* $u \in \mathcal{U}_f$*. It maintains a list of values* X *which is initially empty, and the current accumulator value* v*, which is initially* u*. It responds to two types of messages: in response to the ("ADD",* x*) message, it checks that* $x \in \mathcal{X}_k$*, and if so, adds* x *to the list* X *and modifies* v *by evaluating* f*, it then sends back this updated value; similarly, in response to the ("DEL",* x*) message, it checks that* $x \in X$*, and if so, deletes it from the list and updates* v *by running* D *and sends back the updated value. In the end of the computation,* M *outputs*

the current values for X and v. Let $\mathcal{U}_f^{'} \times \mathcal{X}_k^{'}$ denote the domains for which the computational procedure for function $f \in \mathcal{F}_k$ is defined. For all probabilistic polynomial-time adversaries $\mathcal{A}_k$,

$$Pr[((f, aux_f) \leftarrow G(1^k); u \leftarrow \mathcal{U}_f; (x^{'}, w) \leftarrow \mathcal{A}_k(f, U_f, u) \leftrightarrow M(f, aux_f, u) \rightarrow (X, v) : \\ w \in \mathcal{U}_f^{'}; x^{'} \in \mathcal{X}_k^{'}; x^{'} \notin X; f(w, x^{'}) = f(u, X)] = neg(k).$$

5 Dynamic Accumulator Scheme from Bilinear Pairings

This proposed group signature scheme is based on an identity escrow scheme with membership revocation. It uses elliptic curves and bilinear pairings and combines several cryptographic techniques. The model which is constructed and later implemented, is derived from the work of Nguyen [33]. Unlike Nguyen's scheme the presented scheme does not need to update the accumulator after a new member has joined the group. The group manager must only refresh the accumulator value when a member is revoked.
The security of this scheme is based on the q-Strong Diffie-Hellman (q-SDH) assumption, see Definition 2.4.5. This definition was strengthened by Boneh and Boyen [10] from a weaker definition proposed by Mitsunari, Sakai and Kasahara [30].
To implement this model, at least one trusted party is needed. This trusted party can be split into two group administrators, the issuer and the opener. Each user $i \in \mathbb{N}$ who joins the group has a unique identity. First this identity is represented as a PKI certificate and after the join protocol as $a_i \in \mathbb{Z}_r$. Due to the use of dynamic cryptographic accumulators, membership revocation can easily be achieved.

5.1 Features of the Scheme

In this section we highlight some important features the scheme provides.

- **Two authorities.** As suggested in previous works, we separate the authority into two entities, the issuer and the opener. Each has its own key. The issuer can add (with the **Join** protocol) and delete (with the **Revoke** algorithm) members to/from the group. The opener can identify the signer of a signature and produce a proof of this claim. This model provides more security (compared to a single authority) in the face of the possibility that authorities can be dishonest.
- **Three key requirements.** We verify that all defined key requirements, namely anonymity, traceability and non-fameability, are fulfilled. It is taken for granted, that the correctness requirement is verified as well.
- **PKI.** We assure that each group member or potential group member is in possession of a digital certificate issued by a trusted PKI. So each member

has a personal public key, which is certified, and a matching private key, which is kept secret. This is necessary to protect members from framing attacks by a dishonest issuer and to trace them in case of a dispute. The issuing of certificates for potential group members is carried out in the **User PKI Certificate Generation** algorithm.

- **Public verifiable proofs of opening.** If the opener is dishonest the produced results from the **Open** algorithm may accuse innocent members. In order to protect them, the opener must produce a public verifiable proof to any claim, proving that an identity generated a particular signature.
- **Public table for witness updates.** Each member keeps a membership secret key and a membership witness. The membership witness proves that a certain value a_i, associated with the member, is accumulated in the accumulator value. When a member is revoked by the issuer, members must update their witness. The **Update Witness** algorithm uses the public table "accu" to update the membership witness.

5.2 Overview of the Scheme

The described scheme is a tuple $\mathcal{DCA}$ = (**Group Key Generation, User PKI Generation, Join protocol, Signature protocol, Update Witness algorithm, Update Accumulator algorithm, Open algorithm, Judge algorithm, Revoke algorithm**) of PT algorithms.

- **Group key generation** generates the public parameters and the secret keys.
- **User PKI generation** outputs a digital certificate for a prospective member, which is embedded in a PKI.
- **Join protocol** lets a user, in possession of a certificate, join the group and transfers a membership secret key and a membership witness.
- **Signature protocol = (Sign, Verify)** allows a group member to sign a message and to anonymously prove his membership in the group.
- **Update witness algorithm** updates the witness of a member.
- **Update accumulator** transfers the current accumulator value to any user.
- **Open algorithm** can only be executed as a group administrator and destroys the anonymity on a given signature.
- **Judge algorithm** checks if the proof outputted by the **Open** algorithm is correct.
- **Revoke algorithm** can revoke a member if he has violated the policy.

The security requirements are explained in Section 3.4.3 and verified in Section 5.12.
In this section, an overview of the scheme with a chronology of a typical membership cycle is given. First, a group is created and a user retrieves an SSL certificate, which is maintained by the PKI. With this certificate the join protocol is secured and the user, in the following examples named Alice, joins the group. After joining the group, Alice updates her witness to be ready to sign a message. The verifier Bob also updates his local accumulator value, so he can later verify a given signature. In the next step the signature protocol is executed.
When a signature that Alice has created arouses suspicion, Bob is able to send the signature to the opener. The opener opens the signature according to some security policy and also produces a proof. Everyone can verify this proof by calling the judge algorithm. After the opening of the signature, the identified member ID can be used by the issuer to revoke this member.
As mentioned above, both administrative operations should be lying in the hands of different entities. This whole process is illustrated in the figure 5.2.

In the following sections we use the symbols of Figure 5.1 to present an overview of the relevant functions.

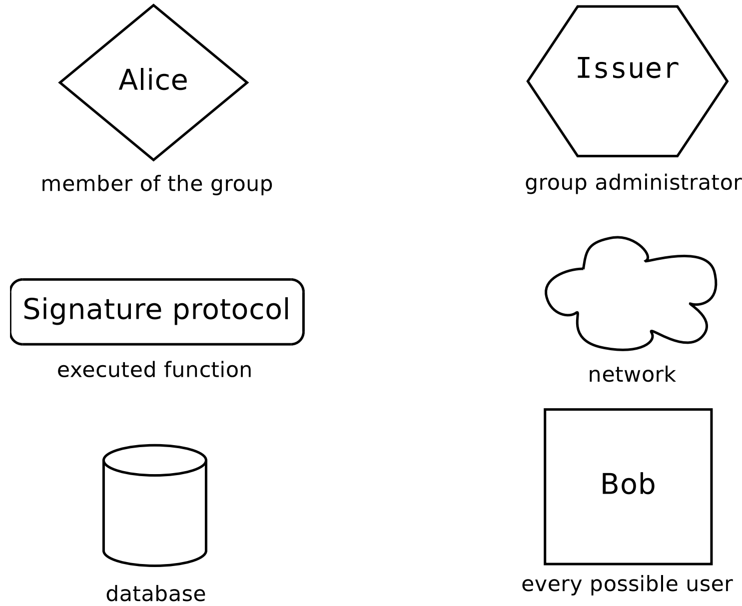


Figure 5.1: Illustration of the different use symbols

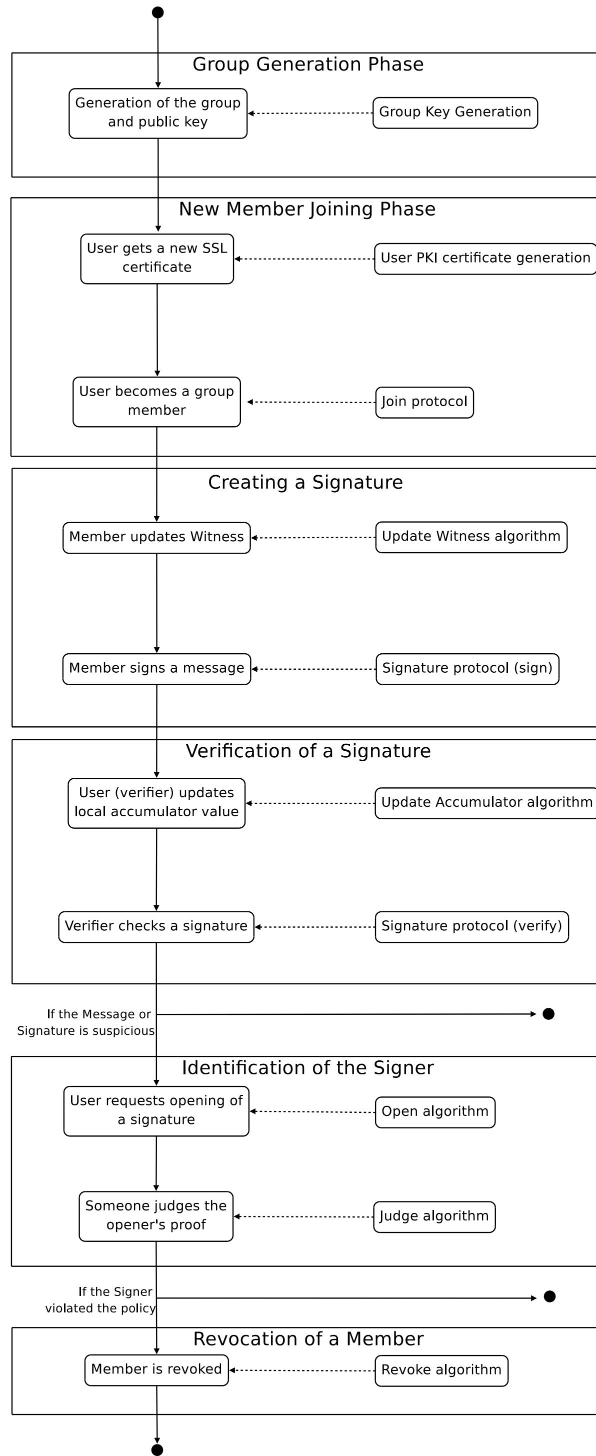


Figure 5.2: Illustration of a chronology overview of the scheme

5.3 Group Key Generation

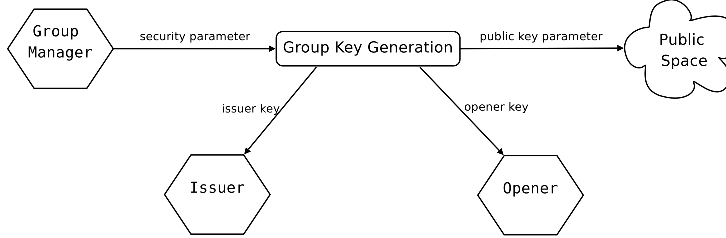


Figure 5.3: Illustration of the group key generation

Inputs:

- r bit: bit length of the group order
- q bit: bit length of the order of the base field

Outputs:

- Groups : all parameters of the groups $\mathbb{Z}_p$, $\mathbb{G}_1$, $\mathbb{G}_T$ and pairing e
- Public group parameter : $(\Sigma, G_2, G_1, G, H, P_{pub}, P_0, P, Q_{pub}, Q, u)$
- Issuer key : (s, x)
- Opener key : $(x^{'})$
- Starting accumulator value : $V_0 = uQ$

Given two security parameters r, q, the function generates a cyclic additive group $\mathbb{G}_1$ (with a prime order of q bits), two multiplicative groups $\mathbb{Z}_p$ (with an order of r bits) and $\mathbb{G}_T$ (with an order of $q * k$ bits, where k is the embedding degree, in this case 2) and a bilinear pairing $e : \mathbb{G}_1 \times \mathbb{G}_1 \rightarrow \mathbb{G}_T$.
After initializing these groups and the pairing, choose
$G, P, G_1, G_2, P_0, H, Q \in_R \mathbb{G}_1,\ s, u, x, x^{'} \in_R \mathbb{Z}_p$ and compute $P_{pub} = xP$, $Q_{pub} = sQ$. The issuer key (x, s) is now perfectly hidden in P_{pub} and Q_{pub}. Compute $\Sigma = e(G, G)^{x^{'}}$ to hide also the opener key. Finally, compute $V_0 = uQ$ to get the first accumulator value. It is not necessary to calculate V_0 here, any random value would fit.

5.4 User PKI Certificate Generation

Inputs:

Figure 5.4: Illustration of the user PKI certificate generation

- CN : unique common name to identify the user
- Password : password to encrypt the private key

Outputs:

- Certificate : a PKI certificate

To identify the user and to ensure a secure connection in the join protocol, a PKI (Public Key Infrastructure) certificate is generated. To generate this certificate, standard software, e.g. OpenSSL, is used. This software must be able to create a certificate which can accomplish both goals, to identify the user and to secure a connection.
It is necessary to save the user's information and the associated common name in a database to be able to provable link a signature to a specific user.
Both parties in the **Join** protocol need a PKI certificate. The user verifies the issuer certificate to be sure that he is connected to the correct server and the issuer authenticates the user on the basis of the user's certificate.

5.5 Join Protocol

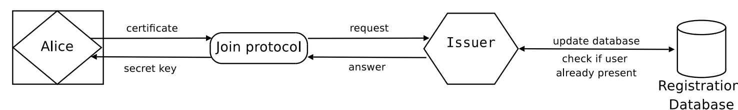


Figure 5.5: Illustration of the join protocol

The protocol is carried out between a user and the issuer.

Inputs:

- Certificate : PKI certificate
- Public group key : public groups and parameter values

- Password : password to encrypt the private values (only user)
- Issuer key : private values (s, x) (only issuer)

User outputs:

- Group secret key : $(x_i, a_i, S_i, \Delta_i)$ encrypted with a password
- User's witness : (j, W_i, V_i)

Issuer outputs:

- Update database : $(\Delta_i, I, u, v, P_i, a_i, S_i, CN,$
  $sign(\mathcal{H}(\Delta_i \mid I \mid P_i \mid a_i \mid S_i))$

In this protocol, a user i joins the group with the authorization of the issuer and gets all needed values to sign anonymously. First of all, a trusted encrypted connection must be established. Due to the use of a PKI, both parties can identify each other and encrypt the connection. After verifying the certificates on both sides, the user checks whether the issuer's common name and the server's DNS name match. The issuer on the other side checks whether the user has already joined the group.
If none of these checks fail, the user and the issuer jointly generate a random value x_i. The issuer does not know the user's secret key x_i but can verify that the random value v and the public value u are contained in it. After user i has proven that all elements of I and x_i are known to him, a unique membership secret key (a_i, S_i, Δ_i) is transferred to the user. The value a_i is randomly chosen, but the issuer must ensure that there is no other user with the same a_i. Finally, user i must sign the values $(\Delta_i, I, P_i, a_i, S_i)$ and gets his membership witness (W_i, V_i). The user is using the secret key that corresponds to the user's PKI certificate. So the user's identity is provable connected to the values $(\Delta_i, I, P_i, a_i, S_i)$. Should actions of the user raise suspicion, the group administrator can present the authenticated user certificate and a provable link between an opened signature and the user's PKI certificate.

1. user $i \rightarrow$ issuer : $I = yP + rH$, where $y, r \in_R \mathbb{Z}_p^*$
2. user $i \leftarrow$ issuer : $v \in_R \mathbb{Z}_p^*$ and current accumulator V_i
3. user i computes : $x_i = uy + v$, $r^{'} = ur$ and $P_i = x_i P$, $H^{'} = r^{'} H$
4. user $i \rightarrow$ issuer : P_i and $H^{'}$ and a proof of knowledge, described in section 5.5.1:
   - user i computes: $t \in \mathbb{Z}_p^*$, $W_1 = tH$, $W_2 = tP$, $c = \mathcal{H}(W_1 \mid W_2 \mid P \mid P_i \mid H^{'} \mid commonname)$, $s_1 = t + cr^{'}$ and $s_2 = t + c_i$
   - user $i \rightarrow$ issuer : W_1, W_2, s_1 and s_2

- issuer computes and verifies whether $c = \mathcal{H}(W_1 \mid W_2 \mid P \mid P_i \mid H^{'}, commonname)$, $vP + uI - P_i = H^{'}$, $s_1H = W_1 + cH^{'}$ and $s_2P = W_2 + cP_i$

5. issuer chooses $a_i \in_R \mathbb{Z}_p^*$ different from all previous a_i and computes $S_i = \frac{1}{a_i+x}(P_i + P_0)$

6. user $i \leftarrow$ issuer : a_i,S_i

7. user i computes : $\Delta_i = e(P, S_i)$ and verifies whether $e(a_iP + P_{pub}, S_i) = e(P, x_iP + P_0)$

8. user $i \leftarrow$ issuer : j number of the current accumulator, $D_i = \frac{1}{a_i+x}(V_j - P_i)$ and the membership witness $W_i = \frac{1}{a_i+s}V_j$

9. user $i \rightarrow$ issuer : verifies $e(a_iP + P_{pub}, D_i) = e(P, V_j - P_i)$ and sends $sign(\mathcal{H}(\Delta_i \mid I \mid P_i \mid a_i \mid S_i))$ ($sign$ is using the private key that corresponds to the user's PKI certificate)

10. issuer computes : $\mathcal{H}(\Delta_i \mid I \mid P_i \mid a_i \mid S_i))$ and $verify(sign(\mathcal{H}(\Delta_i \mid I \mid P_i \mid a_i \mid S_i))$ ($verify$ is using the public key embedded in the user's PKI certificate), if successful the issuer appends the user to the registration database

11. user stores the membership secret key $(x_i, a_i, S_i, \Delta_i)$ (secured with an encryption mechanism) together with the membership witness (W_i, V_j)

5.5.1 Proof of Knowledge

The proof of security of the non-interactive zero-knowledge protocol executed in the join protocol contains three parts.

- **Correctness** To prove correctness, all equations must be correct.

$$\begin{aligned}
H^{'} &= vP + uI - P_i \\
urH &= vP + uyP + urH - x_iP \\
x_iP &= vP + uyP \\
(v + uy)P &= vP + uyP \\
s_1H &= W_1 + cH^{'} \\
(t + cr^{'})H &= tH + cr^{'}H \\
s_2P &= W_2 + cP_i \\
(t + cx_i)P &= tP + cx_iP
\end{aligned}$$

So if both parties are honest, all calculations will be correct. □

- **Soundness** If the discrete logarithm assumption (Definition 2.4.2) holds in $\mathbb{G}_1$, then a prospective member must have knowledge of x_i and $r^{'}$ to convince the issuer with non-negligible probability. Due to the knowledge of x_i and $r^{'}$, also $P_i = x_i P$ and $H^{'} = r^{'} H$ are known. The prospective member must also have knowledge about y and r, otherwise the computation of $I = yP + rH$ is not possible.
  Suppose there are two pairs of challenges and responses (c, y_1, y_2) and $(c^{'}, y_1^{'}, y_1^{'})$ for the same commitment W_1, W_2. With the knowledge of both challenges and responses the verifier can compute

$$
\begin{aligned}
y_1 - y_1^{'} &= cr^{'} - c^{'} r^{'} = r^{'}(c - c^{'}) \\
\Rightarrow r^{'} &= \frac{r^{'}(c - c^{'})}{(c - c^{'})} \\
y_2 - y_2^{'} &= cx_i - c^{'} x_i = x_i(c - c^{'}) \\
\Rightarrow x_i &= \frac{x_i(c - c^{'})}{(c - c^{'})}
\end{aligned}
$$

  Therefore the joiner (prover) must have knowledge of $x_i, r^{'}, y$ and r to convince an honest issuer.
  Basically, if there are two pairs of challenges and responses for one commitment, one can construct a knowledge extractor. □

- **Zero-knowledge** To prove the zero-knowledge property, the hash function is modeled as a random oracle, see Figure 5.8 and the work of Bellare and Rogaway [5] for more information.
  The values t, c, y and r are randomly chosen and $y_1 = t$, $y_2 = t$, $W_1 = tH - cH^{'}$ and $W_2 = tP - cP_i$ are computed. The values $(W_1 \mid W_2 \mid P \mid P_i \mid H^{'} \mid \mathit{commonname})$ and c are entered into the random oracle. If $(W_1 \mid W_2 \mid P \mid P_i \mid H^{'} \mid \mathit{commonname})$ has already been sent to the oracle with $c^{'} \neq c$, another $(W_1 \mid W_2 \mid P \mid P_i \mid H^{'} \mid \mathit{commonname})$ and c will be chosen. At last $I = u^{-1}P_i - vu^{-1}P + u^{-1}H$ is computed and along with (W_1, W_2, y_1, y_2) sent to the issuer. The dishonest verifier, using the random oracle, will not be able to notice a difference between this communication and the real communication. □

5.6 Signature Protocol

This protocol is carried out between a signer and a verifier. Notice that the verifier does not have to be a member of the group.
This protocol aims at showing the signer's knowledge of a secret x_i, (a_i, S_i), and the signature on a specific message. The only information anyone can get from a protocol run, except for the opener, is whether the signer is a member of the group or not. Before the computation is started, the signer checks with

Update Witness whether the membership witness is up-to-date.
The verifier also updates his accumulator value with the program **Update Accumulator** prior to the execution of the verify algorithm. This protocol is non-interactive so the part of the signer and the verifier are described separated.

5.6.1 Sign

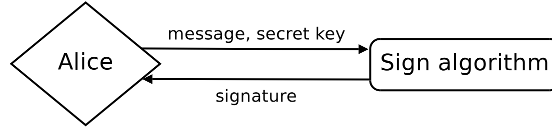


Figure 5.6: Illustration of the sign algorithm

The sign algorithm can only be executed by a group member. It produces a group signature on a given message.
Inputs:

- Message : target message which should be signed
- Public group key : public groups and parameter values
- Password : password to decrypt the private values
- Group secret key : $(x_i, a_i, S_i, \Delta_i)$ encrypted with a password
- User's witness : (j, W_i, V_i)

Outputs:

- Signature :
  $(E, \Lambda, U_1, U_2, R, T_1, T_2, T_3, \Pi_1, \Pi_2, \Pi_3, s_0, s_1, s_2, s_3, s_4, s_5, s_6, s_7, s_8)$

The following steps are performed to produce a valid signature.

1. signer computes random values:
   $t, r_1, r_2, r_3, k_0, k_1, k_2, k_3, k_4, k_5, k_6, k_7, k_8 \in_R \mathbb{Z}_r^*$
2. signer computes ElGamal encryption of Δ_i: $E = tG$, $\Lambda = \Delta_i \Sigma^t$
3. signer computes initial values:
   $U_1 = S_i + r_1 H$,
   $U_2 = W_i + r_2 H$,
   $R = r_1 G_1 + r_2 G_2 + r_3 H$,
   $T_1 = k_1 G_1 + k_2 G_2 + k_3 H$,

$T_2 = k_4G_1 + k_5G_2 + k_6H - k_7R,$
$T_3 = k_8G,$
$\Pi_1 = e(P,P)^{k_0}e(P,U_1)^{-k_7}e(P,H)^{k_4}e(P_{pub},H)^{k_1},$
$\Pi_2 = e(Q,U_2)^{-k_7}e(Q,H)^{k_5}e(Q_{pub},H)^{k_2},$
$\Pi_3 = e(P,H)^{-k_1}\Sigma^{k_8}$

4. signer computes challenge: $c = \mathcal{H}(\text{message} \mid T_3 \mid \Pi_1)$
5. signer computes response: $s_0 = k_0 + cx_i$, $s_1 = k_1 + cr_1$, $s_2 = k_2 + cr_2$, $s_3 = k_3 + cr_3$, $s_4 = k_4 + cr_1a_i$, $s_5 = k_5 + cr_2a_i$, $s_6 = k_6 + cr_3a_i$, $s_7 = k_7 + ca_i$, $s_8 = k_8 + ct$
6. signer outputs: signature $(E, \Lambda, U_1, U_2, R, T_1, T_2, T_3, \Pi_1, \Pi_2, \Pi_3, s_0, s_1, s_2, s_3, s_4, s_5, s_6, s_7, s_8)$

This signature can be verified by the **Verify** algorithm.

5.6.2 Verify

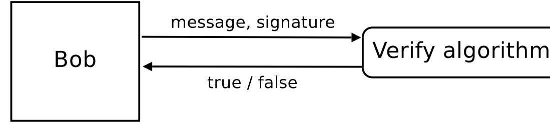


Figure 5.7: Illustration of the verify algorithm

The verify algorithm can be executed anyone, who knows the group public key.
Inputs:

- Message : target message which should be signed
- Public group key : public groups and parameter values
- Current accumulator : V_i

Outputs:

- Result : true or false

The following steps are performed to validate a signature.

1. verifier computes challenge: $c = \mathcal{H}(\text{message} \mid T_3 \mid \Pi_1)$
2. verifier checks that:
$T_1 = s_1G_1 + s_2G_2 + s_3H - cR,$
$T_2 = s_4G_1 + s_5G_2 + s_6H - s_7R,$
$T_3 = s_8G - cE,$

$\Pi_1 = e(P,P)^{s_0} e(P,U_1)^{-s_7} e(P,H)^{s_4} e(P_{pub},H)^{s_1} e(P,P_0)^c e(P_{pub},U_1)^{-c}$,
$\Pi_2 = e(Q,U_2)^{-s_7} e(Q,H)^{s_5} e(Q_{pub},H)^{s_2} e(Q,V_i)^c e(Q_{pub},U_2)^{-c}$,
$\Pi_3 = e(P,H)^{-s_1} \Sigma^{s_8} \Lambda^{-c} e(P,U_1)^c$

3. verifier outputs: true, if all equations are correct or false else

5.6.3 Security Proof of the Underlying Zero-knowledge Protocol

The signature protocol described above is based on a zero-knowledge protocol. To show the security of this non-interactive zero-knowledge signature protocol, three properties (see Section 2.3) must be fulfilled.

- **Correctness** The correctness property is fulfilled if all equations are correct. So an honest signer and an honest verifier can always execute the protocol.
  The first equation T_1 includes the values $c, k_1, k_2, k_3, r_1, r_2$ and r_3.

$$\begin{aligned} T_1 &= s_1G_1 + s_2G_2 + s_3H - cR \\ &= (k_1 + cr_1)G_1 + (k_2 + cr_2)G_2 + (k_3 + cr_3)H - \\ &\quad c(r_1G_1 + r_2G_2 + r_3H) \\ &= k_1G_1 + cr_1G_1 - cr_1G_1 + k_2G_2 + cr_2G_2 - cr_2G_2 + k_3H + \\ &\quad cr_3H + cr_3H - cr_3H \\ &= T_1 \end{aligned}$$

  The values $c, a_i, k_4, k_5, k_6, k_7, r_1, r_2$ and r_3 are included in the equation T_2. So the user's identifier a_i is now linked to the hashed message c.

$$\begin{aligned} T_2 &= s_4G_1 + s_5G_2 + s_6H - s_7R \\ &= (k_4 + cr_1a_i)G_1 + (k_5 + cr_2a_i)G_2 + (k_6 + cr_3a_i)H - \\ &\quad (k_7 + ca_i)(r_1G_1 + r_2G_2 + r_3H) \\ &= k_4G_1 + (cr_1a_i)G_1 - (cr_1a_i)G_1 + k_5G_2 + (cr_2a_i)G_2 - \\ &\quad (cr_2a_i)G_2 + k_6H + (cr_3a_i)H - (cr_3a_i)H - k_7R \\ &= T_2 \end{aligned}$$

  In the equation T_3, a part of the ElGamal encryption E, and therefore t, is linked to the values k_8 and c.

$$\begin{aligned} T_3 &= s_8G - cE \\ &= (k_8 + ct)G - ctG \\ &= k_8G = T_3 \end{aligned}$$

After showing the correctness of the sum equations, the product equations, using pairings and their properties, are reviewed. Π_1 connects the values of $S_i, c, x_i, a_i, k_0, k_1, k_4, k_7$ and r_1, now all sum equations are linked to Π_1 and x_i.
In the last part of the calculation of Π_1, the exponents of different pairings are calculated separately for a better overview.

$$\begin{aligned}
\Pi_1 = {} & e(P,P)^{s_0} e(P,U_1)^{-s_7} e(P,H)^{s_4} e(P_{pub},H)^{s_1} e(P,P_0)^c \\
& e(P_{pub},U_1)^{-c} \\
= {} & e(P,P)^{k_0+cx_i} e(P,U_1)^{-k_7-ca_i} e(P,H)^{k_4+cr_1a_i} \\
& e(P_{pub},H)^{k_1+cr_1} e(P,P_0)^c e(P_{pub},U_1)^{-c} \\
= {} & e(P,P)^{k_0} e(P,U_1)^{-k_7} e(P,H)^{k_4} \\
& e(P_{pub},H)^{k_1} e(P,P)^{cx_i} e(P,U_1)^{-ca_i} e(P,H)^{cr_1a_i} \\
& e(P_{pub},H)^{cr_1} e(P,P_0)^c e(P_{pub},U_1)^{-c} \\
= {} & \Pi_1 \cdot e(P,P)^{cx_i} e(P,S_i + r_1H)^{-ca_i} e(P,H)^{cr_1a_i}) \\
& e(P,H)^{xcr_1} e(P,P_0)^c e(P,S_i + r_1H)^{-cx} \\
= {} & \Pi_1 \cdot e(P,P)^{cx_i} e(P, \frac{1}{a_i+x}(P_i + P_0) + r_1H)^{-ca_i} \\
& e(P,H)^{cr_1a_i} e(P,H)^{xcr_1} e(P,P_0)^c \\
& e(P, \frac{1}{a_i+x}(P_i+P_0) + r_1H)^{-cx} \\
= {} & \Pi_1 \cdot e(P,P)^{cx_i} e(P, \frac{1}{a_i+x}(P_i+P_0))^{-ca_i} e(P,P_0)^c \\
& e(P, \frac{1}{a_i+x}(P_i+P_0))^{-cx} e(P,H)^{-cr_1a_i} e(P,H)^{cr_1a_i} \\
& e(P,H)^{xcr_1} e(P,H)^{-xcr_1} \\
= {} & \Pi_1 \cdot e(P,P)^{cx_i} e(P,P)^{-x_i(\frac{ca_i}{a_i+x})} e(P,P_0)^c \\
& e(P,P_0)^{-\frac{ca_i}{a_i+x}} e(P,P)^{-x_i\frac{cx}{a_i+x}} \\
& e(P,P_0)^{-\frac{cx}{a_i+x}}, \\
& \text{where } cx_i - x_i(\frac{ca_i}{a_i+x}) - x_i(\frac{cx}{a_i+x}) \\
& = cx_i - x_i(\frac{c(a_i+x)}{a_i+x}) = 0 \\
& \text{and } c - \frac{ca_i}{a_i+x} - \frac{cx}{a_i+x} = c - c = 0 \\
\rightarrow {} & \Pi_1 = \Pi_1
\end{aligned}$$

Now the values $W_i, V_i, c, a_i, k_2, k_5, k_6, k_7$ and r_2 are included in the equation for Π_2. This equation checks whether the signer's a_i has been revoked

from the current accumulator value V_i. Note, that the verifier receives V_i from the **Update Accumulator** algorithm. Therefore, the witness of the signer, W_i, must be up-to-date.

$$\begin{aligned}
\Pi_2 &= e(Q,U_2)^{-s_7}e(Q,H)^{s_5}e(Q_{pub},H)^{s_2}e(Q,V_i)^{c}e(Q_{pub},U_2)^{-c}\\
&= e(Q,U_2)^{-k_7}e(Q,H)^{k_5}e(Q_{pub},H)^{k_2}e(Q,U_2)^{-ca_i}e(Q,H)^{cr_2a_i}\\
&\quad e(Q_{pub},H)^{cr_2}e(Q,V_i)^{c}e(Q_{pub},U_2)^{-c}\\
&= \Pi_2\cdot e(Q,W_i+r_2H)^{-ca_i}e(Q,H)^{cr_2a_i}e(Q_{pub},H)^{cr_2}e(Q,V_i)^{c}\\
&\quad e(Q_{pub},W_i+r_2H)^{-c}\\
&= \Pi_2\cdot e(Q,W_i)^{-ca_i}e(Q,r_2H)^{-ca_i}e(Q,H)^{cr_2a_i}e(Q_{pub},H)^{cr_2}\\
&\quad e(Q_{pub},r_2H)^{-c}e(Q,V_i)^{c}e(Q_{pub},W_i)^{-c}\\
&= \Pi_2\cdot e(Q,W_i)^{-c(a_i+s)}e(Q,V_i)^{c}\\
&\quad \text{where } W_i=\frac{1}{a_i+s}V_i\\
&\rightarrow \Pi_2=\Pi_2
\end{aligned}$$

In the last equation, the remaining part of the ElGamal encryption Λ is linked to S_i, c, t, k_1, k_8 and r_1.

$$\begin{aligned}
\Pi_3 &= e(P,H)^{-s_1}\Sigma^{s_8}\Lambda^{-c}e(P,U_1)^{c}\\
&= e(P,H)^{-k_1}\Sigma^{k_8}e(P,H)^{-cr_1}\Sigma^{ct}\Lambda^{-c}e(P,U_1)^{c}\\
&= \Pi_3\cdot e(P,H)^{-cr_1}\Sigma^{ct}\Sigma^{-ct}\Delta_i^{-c}e(P,S_i+r_1H)^{c}\\
&= \Pi_3\cdot e(P,H)^{-cr_1}e(P,H)^{cr_1}\Delta_i^{-c}e(P,S_i)^{c}\\
&\rightarrow \Pi_3=\Pi_3
\end{aligned}$$

So an honest prover and an honest verifier can calculate the equations without any errors. □

- **Soundness** If the discrete logarithm (Definition 2.4.2) holds in $\mathbb{G}_1$, a PPT signer must have the knowledge of $W_i, (a_i, S_i), x_i$ and t such that $e(a_iQ+Q_{pub}, W_i) = e(Q, V_i), e(a_iP + P_{pub}, S_i) = e(P, x_iP + P_0)$, $E = tG$ and $\Lambda = e(P, S_i)\Sigma^t$, so that the verifier in the protocol accepts with non-negligible probability. Suppose there are two pairs of challenges and responses $(c, s_0, \ldots, c_8)$ and $(c^{'}, s_0^{'}, \ldots, s_8^{'})$ for the same commitment $(E, \Lambda, U_1, U_2, R, T_1, T_2, T_3, \Pi_1, \Pi_2, \Pi_3)$.
Let $f_i = \frac{s_i - s_i^{'}}{c-c^{'}}, i = 0, \ldots, 8$, then $f_1 = \frac{k_1+cr_1-k_1-c^{'}r_1}{c-c^{'}}, f_2 = r_2, f_3 = r_3, f_4 = r_1a_i, f_5 = r_2a_i, f_6 = r_3a_i, f_7 = a_i, f_8 = t$ and $f_0 = x_i$, where

$$\begin{aligned}
R &= f_1G_1 + f_2G_2 + f_3H,\\
f_7R &= f_4G_1 + f_5G_2 + f_6H,\\
E &= f_8G,
\end{aligned}$$

$$e(P_{pub}, U_1)e(P, P_0)^{-1} = e(P, P)^{f_0}e(P, U_1)^{-f_7}e(P, H)^{f_4}e(P_{pub}, H)^{f_1},$$
$$e(Q_{pub}, U_2)e(Q, V_i)^{-1} = e(Q, U_2)^{-f_7}e(Q, H)^{f_5}e(Q_{pub}, H)^{f_2},$$
$$\Lambda e(P, U_1)^{-1} = e(P, H)^{-f_1}\Sigma^{f_8}.$$

From the first two equations above, the prover can calculate $\infty = (f_4 - f_7f_1)G_1 + (f_5 - f_7f_2)G_2 + (f_6 + f_7f_3)H$ (∞ is the identity element of $\mathbb{G}_1$, see 2.2.1). The discrete logarithm definition in $\mathbb{G}_1$ implies that $f_4 = f_7f_1$ and $f_5 = f_7f_2$. So the prover has knowledge of $W_i, (a_i, S_i), x_i$ and t satisfying the relations.
Basically, this means that if there are two pairs of challenges and responses for one commitment, one can build a knowledge extractor. □

- **Zero-knowledge** To prove the zero-knowledge property of this non-interactive zero-knowledge protocol, it is necessary to assume that the hash function $\mathcal{H}$ is a random oracle. A dishonest verifier $\mathcal{V}^*$ can ask the random for oracle hash values.
Whenever $\mathcal{V}^*$ asks for a signature of m, the simulator randomly chooses $(E, \Lambda, U_1, U_2, R, c, s_0, \ldots, s_8)$ and computes :

$$T_1 = s_1G_1 + s_2G_2 + s_3H - cR,$$
$$T_2 = s_4G_1 + s_5G_2 + s_6H - s_7R,$$
$$T_3 = s_8G - cE,$$
$$\Pi_1 = e(P, P)^{s_0}e(P, U_1)^{-s_7}e(P, H)^{s_4}e(P_{pub}, H)^{s_1}e(P, P_0)^{c}e(P_{pub}, U_1)^{-c},$$
$$\Pi_2 = e(Q, U_2)^{-s_7}e(Q, H)^{s_5}e(Q_{pub}, H)^{s_2}e(Q, V_i)^{c}e(Q_{pub}, U_2)^{-c},$$
$$\Pi_3 = e(P, H)^{-s_1}\Sigma^{s_8}\Lambda^{-c}e(P, U_1)^{c}.$$

Now (m, T_3, Π_1, c) is entered into the table of the random oracle. If (m, T_3, Π_1) has already been send to the random oracle with $c^{'} \neq c$, another T_3, Π_1 and c will be chosen. Finally, the signature is sent to the verifier. See Figure 5.8 for an overview of the simulation.
The random oracle works as defined below:

1. Given $(m, T_3, \Pi_1) \notin$ *oracletable* choose random $c \in_R \mathbb{Z}_p$, enter (m, T_3, Π_1, c) in the *table* and output c.
2. Given (m, T_3, Π_1) such that $\exists c : (m, T_3, \Pi_1, c) \in$ *arcaletable*, return c.

For the adversary, in this case the dishonest verifier, it is not possible to distinguish a successful communication with the prover from one with the simulator. □

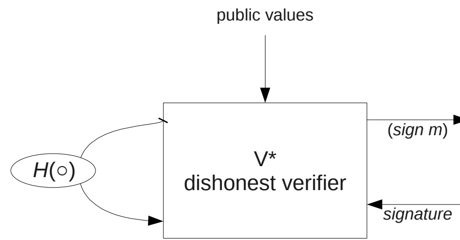


Figure 5.8: Diagram of the simulation in the random oracle model

5.7 Update Witness Algorithm

Inputs:

- Public group key : public groups and parameter values
- Password : password to decrypt the private values
- Group secret key : a_i encrypted with a password
- User's witness : (j, W_i, V_i)

Outputs:

- New user's witness : $(j + n, W_{i+n}, V_{i+n})$

Recall that in the accumulator database, each entry consists of three values (a_i, b, V_i). The value a_i is the identifier of the member that has been revoked. The bit b specifies if the accumulator has been updated in the course of a revocation or another event. And V_i is the current accumulator value. For more information about the database see 6.1.
With an established connection to the accumulator database, a user can update his membership witness. The member retrieves each row from the accumulator database table below j, which was the current accumulator at the last time the member updated his witness.
To calculate the new witness W_{i+n}, the following procedure is executed:

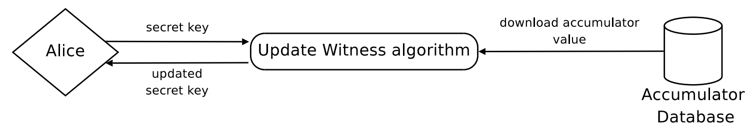


Figure 5.9: Illustration of the update witness algorithm

for $(k = j + 1; k + +; k \leq (j + n))$ **do**

$b_1 = \frac{1}{a_k - a_i}$;

$b_2 = \frac{1}{a_i - a_k}$;

$W_k = b_1 W_{k-1} + b_2 V_{k-1}$;

$V_k = V_{k-1}$;

end for
return $(j + n, W_{i+n})$;

Now the member can sign messages with an up-to-date witness.

5.8 Update Accumulator Algorithm

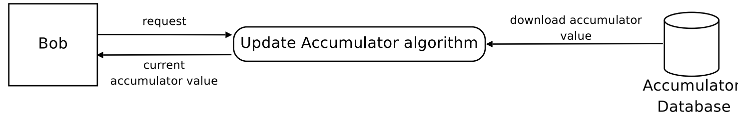


Figure 5.10: Illustration of the update accumulator algorithm

Inputs:

- None : no inputs required

Outputs:

- New accumulator : (V_j)

This algorithm establishes a secure connection to the accumulator database and returns the current accumulator value.

5.9 Open Algorithm

Inputs:

- Public group key : public groups and parameter values
- Opener's secret key : $x^{'}$
- Signature :
  $(E, \Lambda, U_1, U_2, R, T_1, T_2, T_3, \Pi_1, \Pi_2, \Pi_3, s_0, s_1, s_2, s_3, s_4, s_5, s_6, s_7, s_8)$

Outputs:

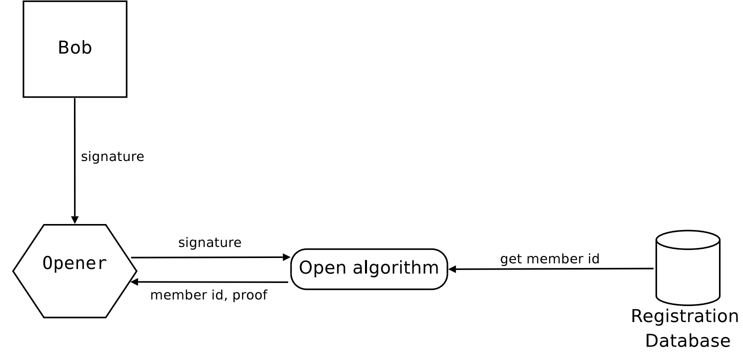


Figure 5.11: Illustration of the open algorithm

- Signing user : $(\Delta_i, I, u, v, P_i, a_i, S_i, CN, sign(\mathcal{H}(\Delta_i \mid I \mid P_i \mid a_i \mid S_i))$
- Non-interactive proof : proof of knowledge of $x^{'}$

The opener can identify the member who has produced the signature. To this end the opener computes $\Delta_i = \Lambda e(E, G)^{x^{'}}$ and a non-interactive proof of knowledge of $x^{'}$. With the resulting Δ_i the user can be uniquely identified in the registration database. In this database, also the signature which was created at the end of the join phase with the member's certificate private key is deposited. So with the proof of the opener and the digital signature, everyone, who posses the public key of the member, can verify that a certain identity has signed a message and authenticate the results produced in the join protocol. The opener proof runs as follows.

- $r_1, r_2 \in_R \mathbb{Z}_r^*$
- $\Gamma_1 = e(G, r_1 G)$ and $\Gamma_2 = e(E, r_2 G)$
- $c_{p1} = \mathcal{H}(\Delta_i \mid \Sigma \mid \Gamma_1 \mid timestamp)$ and $c_{p2} = \mathcal{H}(\Delta_i \mid \frac{\Lambda}{\Delta_i} \mid \Gamma_2 \mid timestamp)$
- $s_{p1} = r_1 - c_{p1} x^{'}$ and $s_{p2} = r_2 - c_{p2} x^{'}$

To verify this proof, the **Judge algorithm** should be executed.

5.10 Judge Algorithm

Inputs:

- Public group key : public groups and parameter values
- Signing user : Δ_i

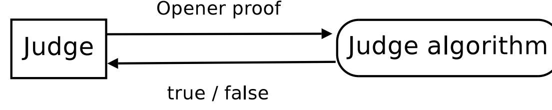


Figure 5.12: Illustration of the judge algorithm

- Non-interactive proof : proof of knowledge of $x^{'}$

Signer Outputs:

- Result : true or false

Anyone who has access to the outputs of the **Open algorithm** can run this algorithm to check whether the proof of knowledge of $x^{'}$ is correct. The check runs as follows:

- check whether $c_{p1} = \mathcal{H}(\Delta_i \mid \Sigma \mid \Gamma_1 \mid timestamp)$ and $c_{p2} = \mathcal{H}(\Delta_i \mid \frac{\Lambda}{\Delta_i} \mid \Gamma_2 \mid timestamp)$
- check whether $e(G, s_{p1}G) = \Gamma_1 \Sigma^{-c_{p1}}$ and $e(E, s_{p2}G) = \Gamma_2(\frac{\Lambda}{\Delta_i})^{-c_{p2}}$

5.10.1 Proof of the Non-interactive Zero-knowledge Protocol

This proof shows that only an opener with the knowledge of the opening key $x^{'}$ is able to convince the judge. To prove this, the three requirements of a zero-knowledge protocol must be fulfilled: correctness, soundness and zero-knowledge.

- **Correctness** The equations given in the judge algorithm must be correct if the opener and the judge are honest. The first check the judge verifies, whether c_{p1} and c_{p2} are the correct hash values, is always true if both parties are honest.
  The following equations are calculated as follows:

$$\begin{aligned} e(G, s_{p1}G) &= \Gamma_1 \Sigma^{-c_{p1}}, \\ e(G, (r_1 - c_{p1}x^{'})G) &= e(G, r_1G)e(G, G)^{-c_{p1}x^{'}}, \\ e(G, G)^{r_1 - c_{p1}x^{'}} &= e(G, G)^{r_1 - c_{p1}x^{'}}, \\ e(E, s_{p2}G) &= \Gamma_2(\frac{\Lambda}{\Delta_i})^{-c_{p2}}, \\ e(E, (r_2 - c_{p2}x^{'})G) &= e(E, r_2G)e(E, G)^{-c_{p2}x^{'}}, \\ e(E, G)^{r_2 - c_{p2}x^{'}} &= e(E, G)^{r_2 - c_{p2}x^{'}}. \end{aligned}$$

It is clear that the correctness requirement is fulfilled. □

- **Soundness** For the soundness requirement the opener is modeled as a Turing machine, so it can be rewound by a cheating judge. Assume the opener can output with non-negligible probability a valid proof without knowing the secret opener key. The judge gets the first zero-knowledge proof from the opener and saves the values $\Gamma_1, \Gamma_2, c_{p1}, c_{p2}, s_{p1}, s_{p2}$. Then the opener is rewound to the point where the hash values c_{p1} and c_{p2} are calculated, note that the time tape of the Turing machine is not rewound. So the hash values change to c'_{p1} and c'_{p2} and thereby s'_{p1} and s'_{p2}. These new values are sent to the judge. Now, the judge solves the following equation:

$$\begin{aligned} s_{p1} &= r_1 - c_{p1}x' \\ s'_{p1} &= r_1 - c'_{p1}x' \\ \Rightarrow s_{p1} - s'_{p1} &= x'(-c'_{p1} + c_{p1}) \\ x' &= \frac{s_{p1} - s'_{p1}}{-c'_{p1} + c_{p1}} \end{aligned}$$

  If there is a way to rewind the opener and produce two pairs of challenges and responses for one commitment, a knowledge extractor can be built that extracts the secret from the cheating opener, what leads to a contradiction. Therefore we conclude that a cheating opener cannot output faked proofs. □

- **Zero-knowledge** To show the zero-knowledge property of the protocol, the random oracle model is used, as presented in Figure 5.8.
  The dishonest opener does not know the secret x'. So the simulator commanding this opener tries to simulate the protocol in a way that an adversary cannot distinguish from a real protocol run. Without knowing the secret, the simulator randomly choses $c_{p1}, c_{p2}, s_{p1}, s_{p2}$. The value Δ_i must be chosen correspondingly to a given signature and Λ. The simulator can use a signature data from an honest opener. Another possibility is that the simulator is a member of the group and generates signatures only to open them by itself.
  The values Γ_1 and Γ_2 are computed as follows:

$$\begin{aligned} \Gamma_1 &= e(G, s_{p1}G)\Sigma^{c_{p1}} \\ \Gamma_2 &= e(E, s_{p2}G)(\frac{\Lambda}{\Delta_i})^{c_{p2}} \end{aligned}$$

  The tuples $(\Delta_i, \Sigma, \Gamma_1, timestamp)$ and $(\Delta_i, \frac{\Lambda}{\Delta_i}, \Gamma_2, timestamp)$ are stored with the corresponding c_{p1} and c_{p2} in the *History* table of the random oracle.
  When an adversary tries to check whether the equations are right, the

results will be correct. The adversary cannot distinguish between a real protocol run and a simulated one. □

5.11 Revoke Algorithm

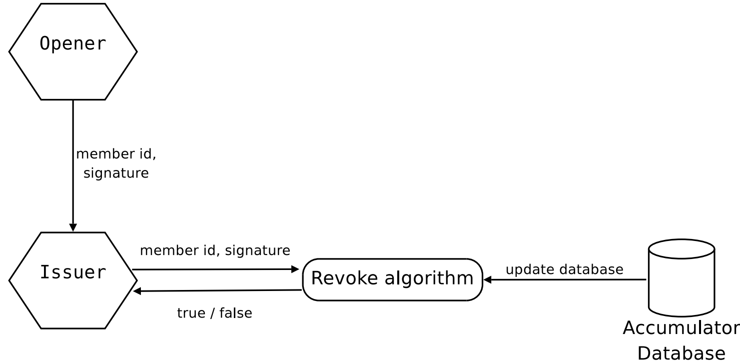


Figure 5.13: Illustration of the revoke algorithm

Inputs:

- Public group key : public groups and parameter values
- Signing user : a_i
- Issuer key : (s, x)

Signer outputs:

- New accumulator value : V_j

This algorithm removes a specific a_i from the accumulator, which in turns transfers to a revocation of the member i from the group. More specifically, the issuer must compute a new accumulator value $V_j = \frac{1}{a_i+s} V_{j-1}$ and store it in the accumulator database. Now every revoked user who wants to call the **Update Witness algorithm** is unable to be successful, since the algorithm as presented would calculate $\frac{1}{a_i - a_i}$. The user's signatures will not be verified successfully if his witness is not up-to-date.

5.12 Proof of the Security Requirements

To execute all following proofs, a new assumption has to be made.

Assumption 5.12.1. *Coalition-Resistance*
If the q-SDH assumption, see 2.4.5, holds, then in this presented scheme, whose group size is bounded by q, the coalition-resistance assumption holds also. The predicate of this assumption $\mathcal{Q}$ is defined as follows:

$$\mathcal{Q}(\langle \Sigma, G_2, G_1, G, H, P, P_{pub}, P_0, Q_{pub}, Q, u\rangle, V_i, \langle x_i, a_i, \Delta_i, S_i\rangle, W_i) = 1$$
$$\Leftrightarrow (e(a_i P + P_{pub}, S_i) = e(P, x_i P + P_0)$$
$$\wedge e(a_i Q + Q_{pub}, W_i) = e(Q, V_i)),$$

where V_i is the current accumulator value.
The proof is only sketched here, for the complete proof see Nguyen [33]. The basic idea is that if a PPT adversary $\mathcal{A}$ can break the coalition-resistance of the scheme, one will be able to construct a PPT adversary $\mathcal{B}$ which can break the q-SDH assumption.
The adversary $\mathcal{A}$ can compute a new membership secret key tuple $(x_i^*, a_i^*, S_i^*, \Delta_i^*, W_i^*)$ which has not been generated before and does not belong to any member in the group. The value $S_i^* = \frac{1}{a_i^*+x}(P_i + P_0)$ and the values P and P_0 can be written as $P = R$ and $P_0 = zR$, where $S_i^* = \frac{1}{a_i^*+x}(x_i^* R + zR)$. If the adversary gets another membership secret key tuple which was generated by an honest user, it will be possible to compute z. When $\mathcal{B}$ gets z it can compute $(c, \frac{1}{c+z}R)$ for any c or in a special case for $c = a_i^* - a_m$, where m is the honest user. Either way the q-SDH assumption does not hold.
As long as the q-SDH assumption holds, the coalition-resistance assumption holds, too. ☐

The definitions of the oracles and the security requirements can be found in Section 3.4.3.

- *Correctness* It is explained above that for the main signature protocol the correctness property holds. One can easily verify, that for all other algorithms and protocols the correctness requirement is also fulfilled. So the scheme has the correctness property. ☐

- *Anonymity* If there is a PPT adversary $\mathcal{A}$ that can break the anonymity property of this scheme, it is possible to construct PPT adversary $\mathcal{B}$ that is able to break the IND-CPA property, see 3.2.2, of the ElGamal encryption. An ElGamal public key (G, Σ) is given and cannot be changed by any of the parties. The adversary $\mathcal{B}$ constructs a complete new group. It generates the public parameters $(\Sigma, G_2, G_1, G, H, P_{pub}, P_0, P, Q_{pub}, Q, u)$ and the issuing key (s, x). The only key unknown to $\mathcal{B}$ is the opening key x', which is also the secret key of the ElGamal encryption. Let $\mathcal{B}$ completely control and simulate the issuer and all possible users of the group. Now $\mathcal{B}$ provides $\mathcal{A}$ with the issuing key and all public values of the group, furthermore $\mathcal{A}$ gains access to following oracles:
  - **SndToI**$(\cdot,\ \cdot)$, **SndToU**$(\cdot,\ \cdot)$, **WReg**$(\cdot,\ \cdot)$, **USK**$(\cdot)$, **CrptU**$(\cdot,\ \cdot)$, **RevokeU**$(\cdot)$ and **Witness**$(\cdot)$. Since $\mathcal{B}$ is in nearly complete control of the group it can easily simulate these oracles.

– **Ch(b,·,·,·)**. $\mathcal{A}$ sends a challenge (i_0, i_1, m) to $\mathcal{B}$. $\mathcal{B}$ finds Δ_{i_b} in the database and has it encrypted by an honest user with the ElGamal algorithm. Now $\mathcal{B}$ simulates the non-interactive zero-knowledge signature protocol as in the proof described in 5.6.3. Then, the complete signature $(E, \Lambda, U_1, U_2, R, T_1, T_2, T_3, \Pi_1, \Pi_2, \Pi_3, s_0, s_1, s_2, s_3, s_4, s_5, s_6, s_7, s_8)$ is sent to $\mathcal{A}$.

Because $\mathcal{A}$ can break the anonymity property, it can send $\mathcal{B}$ the correct identity i_0 or i_1 which has signed the message m. So $\mathcal{B}$ can distinguish between two ciphertexts of an ElGamal encryption with non-negligible probability. As long as the IND-CPA property of the ElGamal encryption holds, the anonymity of the scheme holds. □

- *Traceability* Suppose there is an adversary $\mathcal{A}$ that can break the traceability requirement. Then it is possible to construct an adversary $\mathcal{B}$ which is able to break the coalition-resistance assumption, described in Section 5.12.1, of the scheme.
In this experiment $\mathcal{A}$ signs a message and sends it to an honest verifier. Suppose the signature is valid but the opener cannot trace the identity of the signer, or the opener is able to find the identity but cannot prove this to the judge.
As shown in 5.6.3, the soundness property for the signature protocol holds. Therefore, $\mathcal{B}$ can find W_i, a_i, S_i, x_i and t fulfilling the equations $E = tG, \Lambda = e(P, S_i)\Sigma^t, e(a_iQ + Q_{pub}, W_i) = e(Q, V_i)$ and $e(a_iP + P_{pub}, S_i) = e(P, P_i + P_0)$. So an honest opener can compute $\Delta_i = \Lambda e(E, G)^{x'}$ from the signature.
The issuer should be uncorrupted and there should be no possibility to change the values in the registration database. Also, the adversary is not able to change the keys of the group members (**CrptU(·)** is not available for group members).
This means that if Δ_i cannot be found in the registration database, the adversary $\mathcal{B}$ will produce a new valid tuple (W_i, a_i, S_i, x_i) for the group. This means that the coalition-resistance assumption is broken.
Conversely, if the q-SDH assumption and the traceability property of the scheme hold, the coalition-resistance assumption will hold also. □

- *Non-frameability* To show that the non-frameability property in this scheme holds, suppose there is a PPT adversary $\mathcal{A}$ which can break this property. Then there exists a adversary $\mathcal{B}$ which can break the discrete logarithm assumption over $\mathbb{G}_1$, noted in section 2.4.2. Therefore, $\mathcal{B}$ is given a challenge $(P, P_c = zP)$, where $P \in \mathbb{G}_1$ and $z \in \mathbb{Z}_r$, and $\mathcal{B}$ is successful if it is able to return z.
The adversary $\mathcal{B}$ constructs a new instance of the group by generating $u, s, x, x', d \in_R \mathbb{Z}_r$ and $G, G_1, G_2, H, Q \in_R \mathbb{G}_1$ and sends the group public key $(u, Q, Q_{pub}, P, P_0 = dP, P_{pub} = xP, H, G, G_1, G_2, \Sigma_a = e(G, G)^{x'})$, the

issuing key (x, s) and the opening key $x^{'}$ to the adversary $\mathcal{A}$. $\mathcal{B}$ simulates the whole group and a set of possible users. It chooses an identity i_0 randomly from the set of possible users.
$\mathcal{B}$ gives $\mathcal{A}$ access to the following oracles:

- **SndToU**(i, M_{in}). If $i \neq i_0$, $\mathcal{B}$ just executes the join protocol with an honest user i and sends M_{in} to this user. Whenever $i = i_0$, $\mathcal{B}$ simulates the join protocol and ensures that $P_{i_0} = P^*$. Suppose the membership secret key which is calculate in this protocol is $(x_{i_0}, a_{i_0}, S_{i_0}, \Delta_{i_0})$, where $x_{i_0} = z$ is unknown to $\mathcal{B}$.
- **WReg**$(\cdot,\cdot)$, **GSig**$(\cdot,\cdot)$, **USK**$(\cdot)$, **RevokeU**$(\cdot)$ and **Witness**$(\cdot)$. Due to the unrestricted control of the group and all possible users, $\mathcal{B}$ can easily provide these oracles. The only exception is the call **USK**(i_0), then $\mathcal{B}$ aborts.

If $\mathcal{A}$ is successful with probability ϵ, then the probability that $\mathcal{A}$ impersonates i_0 in the signature protocol is $\frac{\epsilon}{q}$, where q is the maximum number of members in the group.
Due to the soundness property of the signature protocol, $\mathcal{B}$ is able to find $a_{i_1}, S_{i_1}, x_{i_1}$ and t, so that $E_a = tG, \Lambda_a = e(P, S_{i_1})\Sigma_a^t$ and the equation $e(a_{i_1}P + P_{pub}, S_{i_1}) = e(P, x_{i_1}P + P_0)$ is true.
Since the used digital signature scheme, using the PKI certificate, is UNF-CMA, see 3.1.2, it holds that $e(P, S_{i_0}) = e(P, S_{i_1})$ or $S_{i_0} = S_{i_1}$.
So $\frac{1}{a_{i_0}+x}(x_{i_0}P + dP) = \frac{1}{a_{i_1}+x}(x_{i_1}P + dP)$. From this equation $\mathcal{B}$ can compute $z = x_{i_0}$.
Conversely, this implies that if the discrete logarithm assumption holds the non-frameability property will also hold. □

6 Environment and Development

The software solution developed in this thesis consists of several applications. The goal is to use standard software where possible, that is highly available, reliable and, if feasible, open-source. To accomplish this goal the first implementations were completely developed under Linux.
In the following we describe the programming language and the used libraries. In Section 6.2 the applications of the scheme are detailed and the use of these programs is explained. The next two sections deal with different problems which showed up during the implementation. In Section 6.5, the development of a web service for test purposes is described and in Section 6.6 we present our final solution.

In Figure 6.1 a raw overview of the implemented programs is given. On the left hand side the client applications are arranged. All client programs can be executed under Windows or Linux. In the center of the figure, we can find the server applications, running in a Linux environment. On the right hand side are the databases. These databases can be platform independent.
Every program in the figure except the **User PKI certificate generation** and the **Certificate issuance process** receives the public key group information (*groupname*.gpk) and parameters of the group (*groupname*.param) as additional input.
A client does not need all the outlined applications. If a user just want to verify given signatures, the only programs he needs are the **Update Accumulator** program and the **Verify** program. A member of the group definitely needs the programs **User PKI certificate generation** and **Join** to get his membership certificate. The PKI application creates a certificate, binding the user's identity to a public key, and transfers it to this user. Only with this certificate from an independent PKI server, the user can successfully execute the **Join** application and get his membership certificate and his membership witness. Through the use of certificates and SSL, the communication is secured in the join protocol. The PKI server runs the certificate issuance process to generate new certificates for prospective members.
The next client-server-process comprises the **Update Witness** program, **Update Accumulator** program and the **Send Accumulator** process on the issuer server. The process has read access to the accumulator database. The process sends either only the current accumulator value if the **Send Accumulator** program requests it or a number of entries from the accumulator table, specified by the **Update Witness** program. In both cases the connection be-

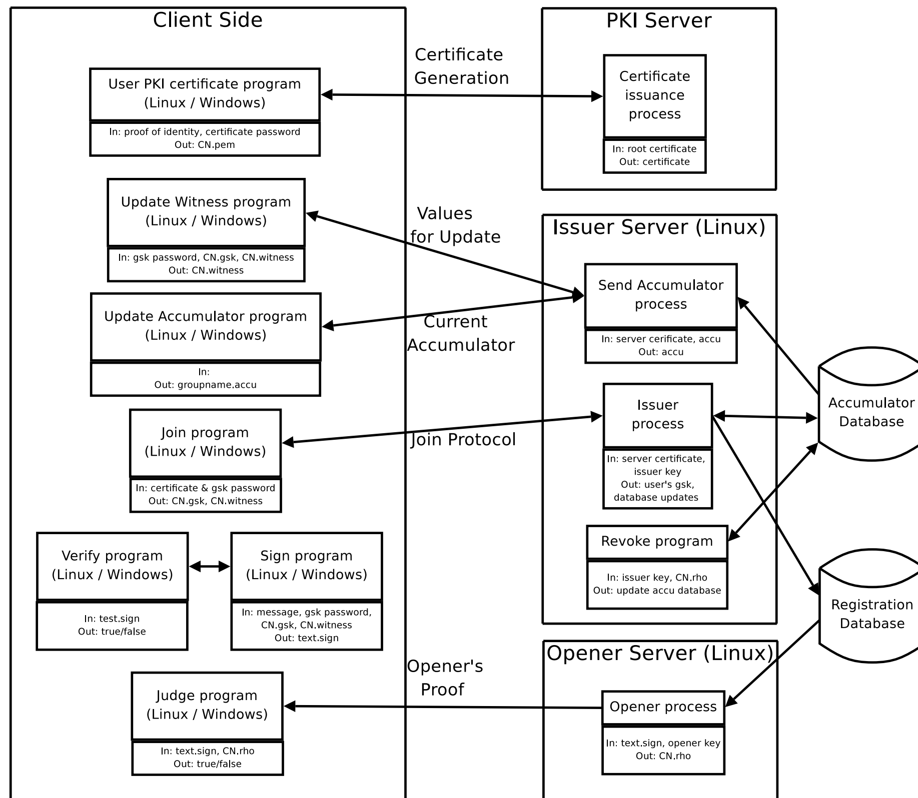


Figure 6.1: Overview of the used programs

tween the **Send Accumulator** process and the program on the client side is secured with the uses of SSL and the certificates created by the PKI.
The programs **Sign** and **Verify** are both executed on the client side and need no input from server processes.
The last client program is the **Judge** program. It checks the proof generated by the **Opener** process. The opener server runs in a Linux environment and is separated from the issuer server to guarantee the separation of the two authorities. The **Revoke** program, which is located on the issuer server, due to the use of the issuing key, gets the needed information to revoke a user from the **Open** process. So both authorities, the issuer and opener server, must agree upon the process of revoking a member.

6.1 Programming Language and Libraries

The programming language is C++, which is a general-purpose programming language, regarded as a middle-level language, because it combines high-level

and low-level language features. There are many C++ compilers and interpreters available, but not all of them are able to compile code under Windows and Linux. After the first development steps, under Linux the compiler *g++* was chosen because it is the standard C++ compiler under Linux. To achieve the same functionality under Windows, the program MinGW (Minimalist GNU for Windows) is used. This is a free native software port of the GNU Compiler Collection (GCC) which is distributed with free import libraries and header files for Windows.
To provide supporting functionalities to the programs of the scheme, several software libraries are used. The PBC (Pairing-Based Cryptography) library [28] is a free C library released under the GNU Public License. The GMP library [21] is a C-library for multiple precision arithmetic (GNU Multiple Precision Arithmetic). The PBC library, is designed to be the backbone of pairing-based implementations and provides such routines like elliptic curve generation, elliptic curve arithmetics and pairing computation. One of the main goals of this library is to provide software portability, but this goal was not easy to accomplish in the implementation. Nonetheless, when working with groups, elliptic curves and pairings, the PBC library is a mighty tool.
There is no adequate implementation of the GMP library for Windows. To solve this problem, the program MSYS (Minimal SYStem) is installed under Windows, which provides a lightweight UNIX-like shell environment to enable the execution of configuration scripts to install UNIX-based programs under Windows. By the use of this shell environment, both libraries (PBC and GMP) could be ported to Windows. After compiling and testing these libraries under Windows the first code parts were transfered.
Another often used library is the OpenSSL (Secure Sockets Layer) library, which does not only implement the SSL and TLS (Transport Layer Security) protocols, but also allows the use of many basic cryptographic and utility functions. The OpenSSL library is also not supported under Windows. After a similar procedure like for the GMP and PBC libraries, the OpenSSL library could be compiled and used under Windows.
All other resources used by this software solution are small libraries which are only one or two files in size. These libraries are not hard to transfer to other operation systems because the complexity of these libraries is small.

6.2 Implementation of the Scheme

6.2.1 User PKI Certificate Generation

This program is a bash script which does not need to be compiled. It is a connection between the Apache web server and the OpenSSL program. When a user registers himself on the web page, his information is first saved in a database, which uses the MySQL (Structured Query Language) system to organize the data, and a new unique serial number for this user is generated.

The bash script calls the OpenSSL functions to generate a new SSL certificate, which stores information about the user. The private key of the certificate, which uses the RSA algorithm, is encrypted with the 3DES (Data Encryption Standard) algorithm. The password for 3DES is chosen by the user.
An example certificate is given below, note that the private key is appended to the certificate to provide all necessary information in just one file for a user. This is not a security risk as long as the password chosen for the encryption of the private key is strong enough. Should an adversary get the encrypted private key the password should resist dictionary attacks, see [37]. The format of the certificate is DER (Distinguished Encoding Rules) which is a message transfer syntax specified by the ITU (International Telecommunication Union) in the document X.960.

```
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number: 14 (0xe)
        Signature Algorithm: sha1WithRSAEncryption
        Issuer: C=DE, ST=NRW, O=Test Co, OU=IT section,
        CN=UserCA/emailAddress=user1@test.com
        Validity
            Not Before: Oct 17 12:34:51 2008 GMT
            Not After : Oct 17 12:34:51 2009 GMT
        Subject: C=DE, ST=NRW, O=TestCooperation, OU=IT,
        CN=user1/emailAddress=user@test.com
        Subject Public Key Info:
            Public Key Algorithm: rsaEncryption
            RSA Public Key: (1024 bit)
                Modulus (1024 bit):
                    00:ab:92:9a:a8:74:b8:24:da:da:ee:de:9a:0c:d1:
                    04:50:10:2a:b2:3b:e2:48:a6:
                Exponent: 65537 (0x10001)
        X509v3 extensions:
            X509v3 Basic Constraints:
                CA:FALSE
            Netscape Comment:
                OpenSSL Client Certificate
            X509v3 Subject Key Identifier:
                01:A6:16:8F:A5:34:D2:34:55:7D:30:50:5E:BF:4C:51:
            X509v3 Authority Key Identifier:
                keyid:49:2E:B7:B8:A8:89:C4:0E:AA:3D:26:BC:30:0E:

    Signature Algorithm: sha1WithRSAEncryption
        dd:97:cc:2c:93:e0:22:b7:81:61:5d:c5:eb:62:8f:75:66:00:
        95:d7:8b:28:00:d2:66:74:49:72:43:37:
-----BEGIN CERTIFICATE-----
```

```
MIIDgTCCAmmgAwIBAgIBDjANBgkqhkiG9w0BAQUFADB+MQswCQYDVQQGEwJERTEM
MAoGA1UECBMDTlJXMRkwFwYDVQQKExBUZXN0IENvb3BlcmF0aW9uMRMwEQYDVQQL
EwpJVCBzZWN0aW9uMRAwDgYDVQQDEwdVc2VyIENBMR8wdP/g==
-----END CERTIFICATE-----
-----BEGIN RSA PRIVATE KEY-----
Proc-Type: 4,ENCRYPTED
DEK-Info: DES-EDE3-CBC,1E39E24712049157

kK43FNPxw/aid4h1xbEv8PQxBpuNO7GBSWmBv42UK+JZsvaQA9zvWF4CMSWNflvc
r7lJGx0ZqpcAwrAvxLMyE7kRhyRgRpXjzg==
-----END RSA PRIVATE KEY-----
```

Subsequent to the registration, the user can download his certificate from the server.
This part of the scheme depends very much on the field of application. In a company or another organized federation there are many more ways to distribute these certificates to the users and to verify a user's identity.

6.2.2 Group Key Generation

The group key generation (GKg) program uses the GMP and PBC libraries besides the standard C++ libraries. This application needs up to four arguments (r, q, d, g). The first two arguments define the bit sizes of the order of the groups $\mathbb{Z}_p$ (bit size of $p = r$) and $\mathbb{G}_1$ (q). First an algorithm searches for two prime numbers with a minimum bit length as given by the arguments. For security reasons, generic discrete logarithm algorithms must be infeasible in groups of order p and finite field discrete logarithm algorithms must be infeasible in finite fields of order q^2. Typical values for p and q are 160 bit and 512 bit.
The other arguments stand for the directory (d) where all generated files are stored and the name of the resulting accumulator group (g).
The first file generated by the function is named *groupname*.param, in which various information about the parameter of the group is saved. In this file one can find the prime numbers which generate $\mathbb{Z}_p$ and $\mathbb{G}_1$ and the summands of the Solinas prime equation which produces r.
The group $\mathbb{G}_T$ is a subgroup of $\mathbb{F}_{q^2}$, thus the embedding degree is 2 and this group is automatically generated whenever a pairing is applied.
Next the program checks whether the generated pairing is symmetric, this ensures a faster pairing but the group elements are larger than in an asymmetric case. After this check, all random elements are initialized and the public group values (also called key) are stored in a file named *groupname*.gpk. The administrator keys are saved in the files *groupname*.ik, issuing key, and *groupname*.ok, opening key.
The last step is saving the first accumulator value in the file *groupname*.accu. This value is used to initialize the MySQL accumulator database. In this database all changes of the accumulator value are recorded and everyone is able to fetch the current accumulator value.

If no error occurs during the execution of the program, the return code is 0; else −1 is returned.
The MySQL accumulator database should abide by some conventions. The accumulator table, called "accu", should look like this:

a_i	*bit*	V_i
0	0	[17787, 764882]
637	1	[2774, 6483]
455	1	[28864, 74630]

Table 6.1: Accumulator Database Scheme

The values in the table above are only examples. The column a_i is the corresponding a_i of the removed user, therefore 0 is an entry in the table when it is initialized. The second column *bit* defines whether a user has been removed (value equals 1) or another event has changed the accumulator value, e.g. initializing the accumulator (then the value equals 0). The last column V_i is the accumulator value, and the last item in this column is the current accumulator.

6.2.3 Join Protocol

The join protocol consists of two parties. The user, who calls the join program, and the group administrator (issuer), who calls the issue program.
The join program uses the GMP and PBC libraries to compute the different values and pairings, the Windows or Linux socket libraries to establish a connection to the issuer, the OpenSSL libraries to secure this connection and use basic cryptographic algorithms like AES, a SimpleIni library to read information from configuration files, and a base64 library to convert simple bits strings to their base 64 representation.
The issue program uses almost the same libraries as the join program, except the MySQL library to connect to the MySQL server, and the syslog library to log messages produced by the program.
Both programs get no arguments but can be configured in their configuration files (Join.conf and Iss.conf).

Issuer
First, the issuer daemon on the server must be running to enable a user to join. To run a program as a daemon, the program must create a child process of itself and then kill itself. Since in Linux no process can run without having a parent process, it is necessary to set the main process in Linux, called init for initialization, as parent process for the issuer daemon. It is also important that this self-created process is not listening to normal control signals like *Ctrl+C* and writes its messages to the syslog daemon, which stores them in */var/log/syslog*.

Only when another process sends the kill signal to a daemon, it will terminate. After the program is running as a daemon, the files *groupname*.param, *groupname*.gpk and *groupname*.ik, created by the group key generation, are loaded. The connection to the MySQL databases and to the TCP/IP [1] socket are established. The network connection is not yet secured.
To secure the connection, a SSL_CTX object is created to provide a framework for all connections this server will handle. This object fetches the server certificate and prepares the relevant resources for the upcoming connections.
Now the issuer daemon enters a never ending while-loop and waits for connection attempts from clients or termination signals from other processes. If a user connects, the connection is only acknowledged if this attempt is a SSL connection. After the SSL handshake, the server checks the certificate of the client. The common name embedded in the certificate identifies the user. So the server can check whether the user has already joined the group, or has been revoked from the group. If one of this status values are set (already joined or revoked), a warning will be output to the user and the connection will be terminated.
The next step is the technical implementation of the overview given in Section 5.5. In step 4, a hash function is used to map the elements $P, P_i, H^{'}$ and CN to one 160 bit string. The issuer uses the public element P, gets the two elements P_i and $H^{'}$ from the joiner and retrieves the CN from the certificate which the user has sent at the beginning of the protocol, before any other elements were transferred. The OpenSSL SHA1 (Secure Hash Algorithm) implementation is used for the hash algorithm.
To choose an unique element a_i for each new user, a function randomly creates an element and then asks the database to ensure that this element has not been created before. If a_i is chosen, the user will now be mapped to a unique element in the group.
At the end of the description given in 5.5, the user sends a signed hash value to the server. To verify these bits, the issuer computes the hash value of the elements Δ_i, I, P_i, a_i and S_i (all are known to the issuer) and then verifies the signed message from the user and compares the result with the output of SHA1($\Delta_i, I, P_i, a_i, S_i$). An algorithm suitable for signing and verifying is RSA. The OpenSSL implementation of this algorithm is used here. In this algorithm the issuer uses the public RSA key of the user to decrypt the signed message. This public key is stored in the certificate of the user, so it does not need to be transfered. After the verification is successful, the issuer updates the database "reg". The database table is constructed as follows:

This table is filled with example values to show how the elements are represented in this database, e.g. S_i is a point on an elliptic curve and is represented as [x,y].
When a new entry has been inserted, the connection is closed and the server waits for a new client, who wants to join the group.

[1] Transport Control Protocol / Internet Protocol

Δ_i	I	u	v	P_i
[162635, 7263655]	[152651, 7636277]	52686	762365	[3516326, 623625]
[162607, 655435]	[565751, 5432156]	76633	863717	[127538, 732098]
[876865, 6854632]	[775535, 9867454]	13244	258098	[405277, 876237]

a_i	S_i	*commonname*	$sign(SHA1(\Delta_i, I, P_i, a_i, S_i))$
73625	[17787, 764882]	user1	5DEF754EAFD
355376	[654437, 6565482]	user2	123D5EEAFD
654315	[987652, 6554221]	user3	6588FEA87B

Table 6.2: Group Registration Database Scheme

Joiner

When the join program is started, it tries to connect to the server (issuer) which is specified in the configuration file. The prospective user also uses the OpenSSL library to create a SSL_CTX object and establishes an encrypted channel over the existing TCP/IP connection. After a secure channel has been created, the client receives a certificate from the server and checks whether the common name in this certificate matches the DNS (Domain Name System) name in the configuration file.

It also be checks if the certificate authority (CA) which signed the server certificate, exists in the file *calist.pem*. This file is used as a database for trusted CAs. In practice, this file could point to a cooperative database where all trusted CAs are registered. Also other checks, like validating the RSA public key against a list of corrupted keys, are possible, but depend heavily on the environment.

The remaining required files are *groupname*.param and *groupname*.gpk. These files are necessary to initialize the mathematical groups and public elements.

If the server does not reject the join query, the prospective user will compute two random values y and r. This operation is platform dependent because in a Linux environment there are sources for random values like */dev/urandom*. Such a source does not exist in a Windows environment, but it is possible to use the RtlGenRandom function [16]. This function establishes a similar functionality like the use of */dev/urandom*, but it is called from the dynamic link library (DLL) *ADVAPI32.DLL*.

The prospective user as well uses the SHA1 and RSA algorithms from the OpenSSL library to compute hashes and sign elements negotiated in the protocol run. The RSA private key is extracted from the user's certificate. The user provides the certificate password, which decrypts the private RSA key, in the configuration file.

When the protocol has finished and no errors occurred, the user saves all negotiated values to different files. To minimize the size of these files, the elements are transformed to their base 64 representation. The base 64 function arranges the input into 6 bit groups and maps each of them to one ASCII [2] character.

[2] American Standard Code for Information Interchange

This action prevents control characters like "space" from showing up.
The so called group secret key (gsk) values are stored in the file *commonname*.gsk, where *commonname* is the common name of the user which is also written in the certificate. The current witness along with the current accumulator value are stored in a separate file *commonname*.witness to enable the **Update Witness** algorithm to access this information.
The secret values in the gsk file are secured by encrypting this file with the AES (Advanced Encryption Standard) algorithm. This algorithm is the successor of DES and it is believed one of the most secure ciphers, today. The user is asked to enter a password, which is at least 8 characters long, to secure his secrets. The password is hashed with the MD5 (Message-Digest algorithm) to generate a key for the AES. The underlying implementation of the AES algorithm was developed by one of the designers of the algorithm.
The new group member can now use this files in the **Signature** protocol.

6.2.4 Signature Protocol

This is a non-interactive protocol because at first one party generates the signature and in the next step, which can be executed at any later point in time and without the involvement of the signer, another party verifies this signature. A Firefox plugin for this protocol exists and is described in detail in Section 6.6.

Signer
There are three arguments a user can give to this program, only two of them must be set. The argument d is optional and informs the program about the directory of the necessary files. These are *groupname*.param, *groupname*.gpk, *commonname*.gsk and *commonname*.witness. Should this argument be absent, the program searches for the files in the same directory in which it is executed. Non-optional are the arguments f and p, which specify the file to sign and the password to decrypt the secret values in *commonname*.gsk. Without these arguments the program fails and throws out an error.
After the secret elements have been successfully decrypted and all values have been loaded, the program generates the necessary random values. Notice that it is again crucial which operating system is used, because on Windows and Linux the correct random functions must be executed.
Now the computations described in Section 5.6, are executed. To map the message to an element in the group $\mathbb{Z}_p$, the whole file containing the message is hashed with SHA1 and then converted to an element.
Before the elements which represent the signature are stored to a file, they are all converted to their base 64 representation to save space. So if the target message is stored in the file *text.txt*, the signature is saved in *text.txt.sign*. These two files are sent to the verifier so he can compute the hash value and verify the given signature.

Verifier
The verify program on the other hand, only needs two arguments. The optional one is again the directory d, and the non-optional is the file containing the message f. Not only the file given as an argument must be in the directory but also the signature file.
To summarize, the required files for this program are: *text.txt*, *text.txt.sign*, *groupname*.param, *groupname*.gpk and *groupname*.accu (in this file the current accumulator value is stored).
After all the elements are initialized and loaded, the message file is hashed and mapped to an element in $\mathbb{Z}_p$. Now all the tests described in the overview in 5.6 are executed and the values $T_1, T_2, T_3, \Pi_1, \Pi_2$ and Π_3 are verified. If all elements pass the tests and no error occurs, the program reports success to the calling environment and the signature is valid.

6.2.5 Update Witness Algorithm

This program is bundled with the sign program. It also uses the configuration file *sign.conf*. Only the signer has a reason to update his witness. Like the verify program, the update witness algorithm has two arguments. First the working directory d to specify the location of the public files, second the password p to decrypt the secret user values. The working directory argument is optional.
First it is checked whether the password has the correct length. If this check fails a -2 will be returned to the calling environment telling the user that he typed in the wrong password.
Now the gsk file is decrypted and all important values are initialized. Then an SSL connection to the accumulator server, which is specified in the configuration file, is established. The common name of the server certificate is checked to ensure that the program is connected to the correct server, and it is checked whether the certificate is authenticated by an authority listed in the *calist.pem* file.
The following computational step is the implementation of the algorithm described in Section 5.7. Every time a new a_k is retrieved from the server, this program checks whether a_k equals the a_i of the user. If this case occurs, an error will be thrown out to inform the user that he is revoked from the group and can no longer update his witness.
If no errors occur, the updated witness will be stored in the file *commonname*.witness or in any other witness file specified in the configuration file.

6.2.6 Update Accumulator Algorithm

This program is bundled with the verify program. It uses the same configuration file *verify.conf*. A verifier can only be certain that a given signature is valid, if he tries to verify it with the current accumulator value. When a group member has been revoked, the value is updated and only then, a verifier with the most

recent value can detect a revoked member.
The only argument which is optional is the working directory d.
The procedure is nearly the same as in the **Update Witness** algorithm. The client connects with a secure SSL connection to the server and verifies the certificate. Then the client gets the current accumulator value and saves it in *groupname*.accu or in any other accumulator file given in the configuration file.

6.2.7 Open Algorithm

This program should only be accessible to group administrators, because it violates the anonymity of the users and needs a ready-only access to the registration database.
Again there are two arguments d and f, whereas the working directory d is optional and f specifies the file containing a signature. Unlike the **Verify** program this algorithm only needs the .sign file. Let us assume that the group administrator has verified the signature with the **Verify** program to be sure that this is a correct and valid signature.
After all public parameters and the opening key (*groupname*.ok) are loaded, the program computes Δ_i by decrypting the EL Gamal encryption. This value is compared to the Δ column in the registration database. If a match is found, the member who has produced the signature will be identified.
A non-interactive zero-knowledge proof of knowledge is produced, to prove that only a group administrator is able to execute the open algorithm . This proof is described in 5.9. The used hash algorithm is again SHA1 of the OpenSSL library.
In the end, a .rho file with information about the member and the proof is created. If there is no match in the Δ column, the file will be named *non*.rho. Such a case is evidence for an error, for example the signature was not from this group or there was an error when the signature was created or transferred.
Normally, this program can match the signature to a member and the file is named *commonname*.rho (*commonname* is the common name of the signing user).

6.2.8 Revoke Algorithm

With this program, a group administrator can revoke a member from the group. To revoke a member, a proof from the open algorithm must be given to the program via the f argument. The working directory argument d is, as always, optional.
In most group structures the opener and the revoker should be different entities to be ensure that not a single group administrator can revoke a member.
After the program has initialized all values from the files *groupname*.param, *groupname*.gpk and *groupname*.ik (the issuing key), the program connects to

the accumulator database. Now it tries to make sure that the given member is not already revoked. Then the program computes a new accumulator value with the calculations shown in Section 5.11.
This new updated accumulator value is stored along with a_i, from the now revoked member i, in the accumulator database.

6.2.9 Judge Algorithm

To verify a proof generated by the **Open** algorithm, the judge algorithm is used. The three arguments are d (optional), $f1$ (the file containing the signature; .sign) and $f2$ (the file containing the proof of knowledge from the opener; .rho).
The algorithm computes the steps described in Section 5.10. If all tests return true, the proof of the opener is valid.
The hash algorithm is SHA1 from the OpenSSL library.

6.3 Compiling on Different Platforms

The execution of a compiler, like **g++**, produces an executable file. On Linux, most of the programs are not pre-compiled when a user gets them. Just all source files and libraries are provided for a user along with a so called **configure** shell script. This script tries to find all platform dependent variables, e.g. if the processor is able to compute 64 bit operations. So this script generates a **Makefile** in which all gathered information are stored. Basically, this **Makefile** consists of all compiler calls which have to be made during the compiling phase, and the correct flags for these calls.
There are several problems with a **configure** script. A major problem is that these scripts are not executable on an operating system like Windows. Another one is their generation; they are generated by a program named **GNU Autoconf**, which is not very user friendly.
For **Autoconf**, a script *configure.ac* and templates *Makefile.in* must be written first. When these files are provided, **Autoconf** constructs the script **configure**. Should everything execute without errors, this procedure is not problematic, but if errors occur, it is not easy to find them. For example, just the **configure** script of the PBC library is about 24000 columns of code.
So we take another approach to the compiling problem. We found the program **CMake** [26] and tested it. It allows a programmer to easily configure and produce a **Makefile**. It does not only provide platform independence, but also compiler independence. So a user can try to compile the code which was written for this thesis and tested with **g++**, with **Visual Studio C++**. While testing the **CMake** program the **Visual Studio C++** compiler was working as expected, but it is not possible to guarantee that every compiler, in any version

on any particular platform can compile this code without any modifications.
CMake reads in a very simple configuration file called **CMakeLists.txt** which must lie in the same directory as the source code. After writing this file, a programmer can run the program **CMake**. It shows graphically whether there are any errors or warnings left, which should be corrected. This program runs under the following platforms: Windows, Mac OSX, Linux, SunOS Sparc, IRIX64, HPUX 9000/785 and AIX powerpc.
CMake builds a **Makefile** for a native environment of the target platform, as shown in figure 6.2.

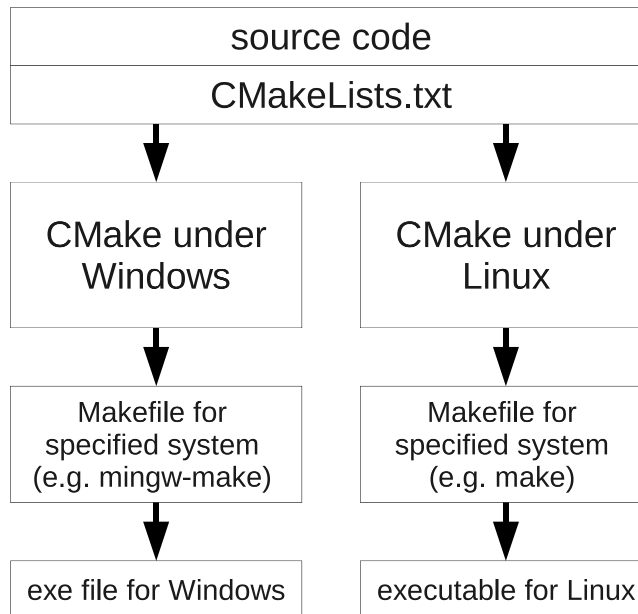


Figure 6.2: Raw description of the program CMake

Nevertheless, the code of this thesis is not able to run on all above mentioned platforms, because there are code parts which are platform dependent. Especially, the random functions and the TCP connection processes are very platform dependent. Also all libraries which are used in the code must be transferred to the target platforms.
For example, the function *tcp_connect* creates a **Winsock** object on Windows to store all necessary information in it. In a Linux environment the same information is stored in a *struct* structure.

6.4 Coding Problems

One of the first solved problems was to find the right library for the pairing functionality. There are many libraries to compute big integers (numbers bigger than 2^{64}) available, but these libraries are almost unable to construct fast pairings or at least well suited elliptic curve groups. The only library, which satisfies all needs of this thesis, is the PBC library, which was coded by Ben Lynn. He also wrote his dissertation [29] about the implementation of pairings. In the next step, the computed elements must be saved in different files and a representation for the elements must be selected. The library provides different functions for this. The elements can be printed as human readable numbers or as a bit string. The bit string is shorter but it can contain different control characters which can make it impossible to read the elements from a file. Thus, the elements are usually saved as a human readable string, except in cases where size matters. In these cases the bit strings are converted to a base 64 representation and saved to files.
The use of human readable integers is not unproblematic. A point on an elliptic curve is represented as *[x-coordinate, y-coordinate]*. The space, the comma and the brackets must be written to the file and be read again. The space character is interpreted as a control character in most of the functions which operate with files. So it makes a difference if an elliptic curve point or a positive number of the group $\mathbb{Z}_p$ is read by the program.
In the first approach, all data was stored in files and a new data handle layer was written, to support this structure. Despite this effort the data storage of all accumulator values and the complete member registration data was not efficient enough. Therefore a database server, in this case the open source MySQL solution, is used. The MySQL solution does not only provide a database server but also a C++ API (Application Programming Interface) which can be used to communicate with the database. This is a high-performance way to store the produced data, besides through using a database solution many more advantageous interesting possibilities arise, like backups and dedicated servers.
After clarifying the data storage, a way to construct secure network connections had to be found. The choice was made to take the obvious solution, the OpenSSL library. It is not easy to write a multi threading OpenSSL server, so that it can check certificates from users and manage more than one connection at a time. Again, performance is still in focus of the project. Because the OpenSSL library, unlike the MySQL library, must be used by both, the client and the server, it has to be transferred to Windows. An SSL connection needs an existing TCP connection, which is platform dependent, so the SSL connection can use the TCP connection to build up a secure channel over it.
Another problem was the hash function for the zero-knowledge proofs. The choice was made to use SHA1 because it produces exactly 160 bit output, which is the same size as an element in the group $\mathbb{Z}_p$, and this algorithm is widely used. The input values of the hash function must be formated correctly. If the wrong size of memory is allocated to store this input, a random value, which is still in the memory, affects the output of the hash function. This means that such an

output cannot be verified on the other side of a connection and is therefore not suitable. When the input problem was solved, the output had to be formatted in such a way, that another function could map it to an element of the group $\mathbb{Z}_p$. The hexadecimal system is used for this transformation.
The next step is the use of configuration files. First a library combined with a shared object file was used for this task, but the transformation to Windows was too complicated, and the library was too large to be fast. The solution is the library SimpleIni which is a small but powerful C++ library for configuration files. The advantage of configuration files is that the user can enter information into these files and does not need to provide the same information, as command line arguments, every time the program is executed. Another possibility to avoid command line arguments, is the definition of variables in the program code. This means that after a change of the variable the program must be recompiled. The last alternative is the fastest one and maybe, in some use cases, this alternative is best suited.

6.5 Integrating a Test Portal

When the different programs were ready, an Apache web server was installed on the server system. This web server operates as a distribution center for the accumulator scheme. New members can get their certificate and the required programs from this web service.
It is important that this service can only be accessed over a secured channel to assert that nobody can eavesdrop this channel and steal information. The OpenSSL architecture, already installed on this server, can be used for the web service as well. The web pages were now only accessible over the port 443 instead of 80. When a user retrieves a page from the server, the channel is secured by the server certificate and the server identifies itself with this certificate.
In this test case the server uses PHP (PHP: Hypertext Preprocessor), a script language executed on the server. After the user enters his identification, the **User PKI certificate generation** is called with the aid of a PHP function.
In a cooperation or other organization, there may be better ways to distribute the certificates to each user.
The web server also provides the option of verifying a signature directly on the website without downloading any programs. It is also possible to open and revoke a given signature with this online service. All of these services are written in PHP and they are formatting the given input and executing the C++ functions described in Section 6.2. No new executables are created for these tasks, so the chance of any incompatibilities between different versions of the program are minimized. In practice, the open and revoke functionality of such a website should be protected by an identification system, so that only group administrators can access these functions.

6.6 Graphical User Interface

To provide a graphical way to sign messages and verify signatures, two Firefox plugins were created. One plugin is able to sign messages and the other one is able to verify signatures. Both plugins are just accessing the executables created before. There is no difference between signatures that are verified with the plugin, on the command line or with the web service on the server. All three variants are producing the same results.
The drawback of GUIs (Graphical User Interfaces) is platform dependency. There are projects which try to reach platform independence, but most of the times the user must install a huge packet in order to run the program. So the decision was made to use the Firefox platform due to the compatibility to most popular platforms, and the supply of a runtime environment for the plugins. Firefox plugins are written in two different programming languages, a description language and a script language. The description language is XUL (XML (Extensible Markup Language) User Interface Language), see also [12; 22]. It describes the design and the graphical layout of the program. Due to the Gecko engine, which Mozilla Firefox and other Mozilla programs are using to generated their design, the layout of the plugin has the same look on every platform. Besides, the user can also change the appearance of his Mozilla product, e.g. using a different theme, without risking to encounter incompatible plugins. XUL is flexible enough to change the graphical layout without changing the functionality.
XUL files are not compiled, like C++ files, but only interpreted by the Gecko engine whenever the plugin is called. Through the use of XML, the XUL files consist of human readable text and can be edited with almost any text editor available. So the plugin can be changed quickly.
The script language is JavaScript, a language developed for web pages and the execution on the client's side. Besides PHP, which is executed on a server every time a web page is requested by the client, JavaScript code is downloaded and executed by the client. This script language is unrelated to Java, a programming language by Sun Microsystems, although Sun Microsystems has trademarked JavaScript.
When programming a plugin with JavaScript, the same syntax and commands can be used as in HTML pages. But a plugin in the Mozilla environment is much more powerful than a script from a website because there are special JavaScript functions available, like file operations or commands that allow to execute local files. All these operations are forbidden for scripts from a website.
The JavaScript, just as XUL, is not compiled but interpreted at the time of execution. One big drawback of this architecture is the performance. It is much slower than executing the files on the command line because the program code is interpreted on-the-fly. An overview is shown in Figure 6.3.

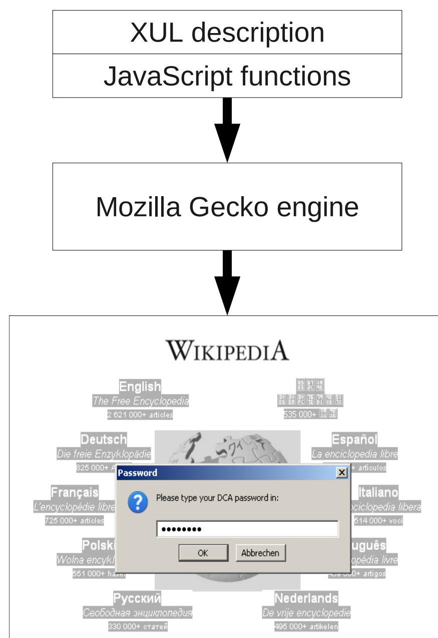


Figure 6.3: Overview of the Mozilla plugin scheme

A problem we encountered concerns the working directory of the plugin. If the plugin has been installed correctly, it is usually located in a subdirectory, often called something like “application data” in the user’s directory. But when the plugin is executed the working directory is the directory where Firefox is installed. So if the plugin wants to use any files, for example configuration files, it must change the directory. For this purpose all compiled programs obtained the argument *-d* to specify the working directory. With the help of JavaScript, the plugin can find its directory on the file system, and provide this information to all programs it calls.
In a use case, there are definitely other ways to provide a user with a graphical user interface. For example, if all possible users of a group use the Windows operating system, the programs could be easily given a GUI with the use of the Visual Studio environment from Microsoft.

7 | Future Work

The proposed implementation in this thesis should be tested in more environments and could be implemented in many more programming languages to provide a better compatibility. Some systems, like Mac, and also architectures, like a 64 bit processor, are not supported yet. But the implementation was tested on a 64 bit processor and due to the Linux compatibility it should also run on a Mac without many changes, although the implementation could be optimized for such conditions.
Maybe it would also be possible to implement this scheme on small processors like micro controllers. The groups for the pairing could be chosen small enough and a fast pairing based library must be supported on the particular controller. In some environments, like car security, the security requirements of the presented scheme are probably interesting.
Also, the implementation as a Firefox plugin may not be the right way to reach most of the users in a normal network. Many users use the Internet Explorer to browse the web. So a small application with high usability could be the better solution in some cases. A platform independent graphical user interface could be constructed by other software solutions, like Qt [1]. With such solutions, the changes in the implementation, by a platform transformation, should be minimal but there will be many problems with speed and layout of the resulting program.
The software library PBC which is used for all pairing based calculations is one of the first libraries capable of this pairing functionality. Pairing based cryptography in general is not widely used. So it is possible that in the near future, with the upcoming of more pairing based algorithms, new faster applications and libraries appear and this implementation could be improved with these new ideas.
Maybe it is also possible to construct a new group encryption scheme building up on this scheme. The public accumulator value could be the public key and only a member of the group, who has not been revoked and therefore has a valid witness, could decrypt a given ciphertext. For example, a user could encrypt information and be sure that only a specific group of people, e.g. Doctors, can decrypt and read the message.

8 Conclusion

The goal of this thesis was the implementation of a dynamic accumulator cryptography system based on pairings and zero-knowledge proofs. One main aspect of the implemented scheme is the independence of calculations linked to the number of members in the group or revoked members. With this implementation it is possible to let any number of users join the group without changing the accumulator value or the group public key. Only the revocation of a member will change that value. The implementation is predominantly platform independent and the parts which are currently implemented only for one operating system can be easily migrated to almost any other platform.
In the Appendix B we show that our group signature can compare in size with a standard digital signature. Also the Appendix C illustrates a speed comparison between our scheme and a standard digital signature scheme. Although our scheme is slower than a digital siganture scheme, the execution times are definitely fast enough to use our scheme in practice.
Another point is that the scheme is well distributable. The server which distributes the accumulator value can be positioned in a network like any other database server, as long as this database supports some kind of replication modes. Just as well as the accumulator, the prospective user and the opener can be positioned in a network. All of these administrative positions can be observed to guarantee the trustworthiness.

Bibliography

[1] Nokia Corporation and/or its subsidiaries. Qt Cross-Platform Application Framework. `http://trolltech.com/products/qt/`, 2008. [Online; accessed 14-12-2008].

[2] Giuseppe Ateniese, Jan Camenisch, Marc Joye, and Gene Tsudik. A Practical and Provably Secure Coalition-Resistant Group Signature Scheme. In *CRYPTO '00: Proceedings of the 20th Annual International Cryptology Conference on Advances in Cryptology*, pages 255–270, London, UK, 2000. Springer-Verlag.

[3] Niko Bari and Birgit Pfitzmann. Collision-Free Accumulators and Fail-Stop Signature Schemes Without Trees. In *EUROCRYPT*, pages 480–494, 1997.

[4] Mihir Bellare, Daniele Micciancio, and Bogdan Warinschi. Foundations of group signatures: Formal definitions, simplified requirements, and a construction based on general assumptions. In *Proceedings of Eurocrypt 2003, volume 2656 of LNCS*, pages 614–629. Springer-Verlag, 2003.

[5] Mihir Bellare and Phillip Rogaway. Random oracles are practical: a paradigm for designing efficient protocols. pages 62–73. ACM Press, 1993.

[6] Mihir Bellare, Haixia Shi, and Chong Zhang. Foundations of group signatures: The case of dynamic groups. In *In proceedings of CT-RSA 05, LNCS series*, pages 136–153. Springer-Verlag, 2005.

[7] Josh Cohen Benaloh and Michael de Mare. One-Way Accumulators: A Decentralized Alternative to Digital Signatures (Extended Abstract). In *EUROCRYPT*, pages 274–285, 1993.

[8] Albrecht Beutelspacher, Jörg Schwenk, and Klaus-Dieter Wolfenstetter. *Moderne Verfahren der Kryptographie. Von RSA zu Zero-Knowledge.* Vieweg, 2006.

[9] Dan Boneh and Xavier Boyen. Efficient Selective-ID Secure Identity-Based Encryption Without Random Oracles. In *EUROCRYPT*, pages 223–238, 2004.

[10] Dan Boneh and Xavier Boyen. Short Signatures without Random Oracles. In *Advances in Cryptology - EUROCRYPT 2004*, pages 56–73. Springer-Verlag, 2004.

[11] Dan Boneh, Emily Shen, and Brent Waters. Strongly unforgeable signatures based on computational Diffie-Hellman. In *In Proc. of PKC 2006*, pages 229–240. Springer-Verlag, 2006.

[12] David Boswell, Brian King, Eric Murphy, Ian Oescheger, and Pete Collins. *Creating Applications with Mozilla*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2002.

[13] Jan Camenisch. *Group Signature Schemes and Payment Systems Based on the Discrete Logarithm Problem*. PhD thesis, ETH Zurich, 1998. Reprint as vol. 2 of *ETH Series in Information Security and Cryptography*, ISBN 3-89649-286-1, Hartung-Gorre Verlag, Konstanz, 1998.

[14] Jan Camenisch and Anna Lysyanskaya. Dynamic Accumulators and Application to Efficient Revocation of Anonymous Credentials. In *CRYPTO '02: Proceedings of the 22nd Annual International Cryptology Conference on Advances in Cryptology*, pages 61–76, London, UK, 2002. Springer-Verlag.

[15] David Chaum and Eugène van Heyst. Group Signatures. In *EUROCRYPT*, pages 257–265, 1991.

[16] Microsoft Corporation. RtlGenRandom Function(Windows). `http://msdn.microsoft.com/en-us/library/aa387694(VS.85).aspx`, 2008. [Online; accessed 14-12-2008].

[17] W. Diffie and M.E. Hellman. Special Feature Exhaustive Cryptanalysis of the NBS Data Encryption Standard. *Computer*, 10(6):74–84, 1977.

[18] Whitfield Diffie and Martin E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, IT-22(6):644–654, 1976.

[19] Nelly Fazio and Antonio Nicolosi. Cryptographic Accumulators: Definitions, Constructions and Applications, 2002.

[20] Amos Fiat and Adi Shamir. How to prove yourself: practical solutions to identification and signature problems. In *Proceedings on Advances in cryptology—CRYPTO '86*, pages 186–194, London, UK, 1987. Springer-Verlag.

[21] Free Software Foundation. The GNU MP Bignum Library. `http://gmplib.org`, 2008. [Online; accessed 14-12-2008].

[22] Mozilla Foundation. XUL -MDC. `https://developer.mozilla.org/en/XUL`, 2008. [Online; accessed 14-12-2008].

[23] Taher El Gamal. A public key cryptosystem and a signature scheme based on discrete logarithms. In *Proceedings of CRYPTO 84 on Advances in cryptology*, pages 10–18, New York, NY, USA, 1985. Springer-Verlag New York, Inc.

[24] Darrel Hankerson, Alfred J. Menezes, and Scott Vanstone. *Guide to Elliptic Curve Cryptography*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2003.

[25] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison Wesley, 2nd edition, November 2000.

[26] Kitware Inc. CMake - Cross Platform Make. `http://cmake.org`, 2008. [Online; accessed 14-12-2008].

[27] Canonical Ltd. Ubuntu Home Page. `http://www.ubuntu.com`, 2008. [Online; accessed 14-12-2008].

[28] Ben Lynn. PBC Library - Pairing-Based Cryptography. `http://crypto.stanford.edu/pbc/`, 2008. [Online; accessed 14-12-2008].

[29] Ben Lynn. On the Implementation of Pairing-Based Cryptography, 2009.

[30] Shigeo MITSUNARI, Ryuichi SAKAI, and Masao KASAHARA. A New Traitor Tracing. *IEICE transactions on fundamentals of electronics, communications and computer sciences*, 85(2):481–484, 2002.

[31] Roger M. Needham and Michael D. Schroeder. Using encryption for authentication in large networks of computers. *Commun. ACM*, 21(12):993–999, 1978.

[32] C. Neuman, T. Yu, S. Hartman, and K. Raeburn. The Kerberos Network Authentication Service (V5). RFC 4120 (Proposed Standard), July 2005. Updated by RFCs 4537, 5021.

[33] Lan Nguyen. Accumulators from Bilinear Pairings and Applications. In *CT-RSA*, pages 275–292, 2005.

[34] Dawn Song and Gene Tsudik. Quasi-efficient revocation of group signatures. In *Proceedings of Financial Cryptography 2002*, pages 183–197. Springer-Verlag, 2002.

[35] Wikipedia the free encyclopedia. A5/1 - Wikipedia. `http://en.wikipedia.org/wiki/A5/1`, 2008. [Online; accessed 14-12-2008].

[36] Wikipedia the free encyclopedia. Digital Signature Algorithm - Wikipedia. `http://en.wikipedia.org/wiki/Digital_Signature_Algorithm`, 2008. [Online; accessed 14-12-2008].

[37] Wikipedia the free encyclopedia. Dictionary attack - Wikipedia. http://en.wikipedia.org/wiki/Dictionary_attack, 2009. [Online; accessed 11-03-2009].

[38] Inc. VeriSign. VeriSign - Security (SSL Certificates), Intelligent Communications, Domain Name Services, and Identity Protection. http://www.verisign.com, 2008. [Online; accessed 14-12-2008].

A | Installation Procedure

In this section we give an overview over the installation of all programs constructed this scheme. First the installation of the server, which is a Linux based environment, is discussed. Note that most of the group administrator's tasks should be distributed to more than one machine and person. Next the installation in a Windows environment is given, so that users can use these programs, for example the Firefox plugin. All given examples have been tested and work as intended. But keep in mind that this is only a test configuration and could be implemented on other platforms if the field of application demands this.

A.1 Linux Installation

This subsection describes installing Ubuntu Linux [27] on a machine. The needed packages are given in brackets after the name of the program.

1. Download and install a Ubuntu version. In a commercial environment the LTS (Long Time Support) version is a good choice, so that the operation system is supported with long time updates.
   Install the desktop version if a graphical user interface is needed for the administrators.

2. After the system is updated, install the OpenSSL (openssl, libssl), GMP (libgmp or [21]), and PBC [28] library. The PBC library must be downloaded from the project site and be installed by a typical install chain (./configure, make, sudo make install).

3. Also the CMake [26] program must be installed on the platform. The necessary package is cmake.

A.1.1 Server Installation

Next, the required server applications are installed. Most of these applications depend on many packages but the package managers of most Linux operating systems resolve these dependencies without any problem. In this example, the Ubuntu package manager *apt-get* tries to find all necessary packages.

1. The first server is the MySQL server (mysql-server). It will automatically install about 20 depending packages. It is important that only one main

MySQL server exists, so that there is one accumulator value database and one registration database. The MySQL library (libmysqlclient) must also be installed. Now the programs can be compiled and can access the database.

2. Next the Apache web server (apache2) should be installed to provide a platform where users can be registered and signatures can be validated. A server certificate for this web server should be created and the SSL modification must be enabled, so this server listens on port 443. The PHP (libapache2-mod-php5) modification must also be installed if the web service described in Section 6.5 will be used.

3. All mentioned server applications have graphical add-on programs which are very useful for a quick and easy administration.

After we have installed all programs on the Linux machine, the server is ready for the programs. The first application **Group key generation** can be compiled and executed. To compile a application, just extract it into a directory, open a terminal session and execute the commands CMAKE . and CCMAKE . . When all parameters are set correctly, the program can be compiled with the command MAKE. The subdirectory */build* contains the compiled program. After the program **Group key generation** is compiled, the group manager can construct the group and all its values.
The next programs which are located on the server, are **Issuer** and **Send accumulator**. The compiling steps are similar to those described above, in the **Group key generation** application. Both are daemons and can be started at the startup of the operating system. The **Issuer** daemon integrates a new user into the group and updates the necessary MySQL databases. The **Send accumulator** daemon sends the current accumulator value to a user or all necessary values to a member who updates his witness.
The issuing server needs the issuing key which is also important for the **Revoke** program. So both applications should be installed on the same server and the MySQL databases must be accessible.
The **Open** program should be installed on a different server to split the administrative operations and the opening and issuing key. But the **Open** program must be able to connect to the registration database.
The **Sign** and **Verify** applications can be compiled under Windows and Linux. In a Windows environment the program CMake must be installed and a graphical user interface guides the user during the process of making a makefile.
The **Judge** program can be installed on any platform.

A.2 Firefox Plugin Installation

To install a Firefox plugin just drag and drop the *.xpi* file into the Mozilla application. After the plugin is installed the configuration files (sign.conf or

verify.conf) should be modified by the user. Also the user files *.gsk* and *.witness* must be copied to the folder of the plugin. Depending on the platform the folder can be found in different locations:

```
Windows:
C:\Documents and Settings\<Windows login/user name>\
Application Data\Mozilla\Profiles\<Firefox user name>
\extensions\<folder of the plugin>\,

Linux (Ubuntu):
/home/<Linux login/user name>/.mozilla/firefox/
<Firefox user name>/extensions/<folder of the plugin>/.
```

After the plugin has been installed and configured, the user can select any kind of text in the Firefox application and press the right mouse button. The context menu shows the operations sign or verify or both. If the user selects one operation the program guides him through the process.
In the subfolder */Content* the program code can be found. The files *main.xul* and *dialog.xul* describe the graphical layout of the plugin. The files *main.js* and *mydialog.js* contain the javascript code, which is executed and calls the appropriate function.

B Space Comparison with PKI-based Certificates

In this chapter the size of a normal standard X.509 certificate is compared to the size of our signature scheme. Both mechanisms provide different security requirements.

B.1 X.509 Certificate and Signature

The following certificate was created by the OpenSSL program. The information given in this certificate is only exemplary and is short on purpose.

```
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number: 22 (0x16)
        Signature Algorithm: sha1WithRSAEncryption
        Issuer: C=DE, ST=NRW, O=Test Cooperation, OU=IT section,
         CN=User CA/emailAddress=userCA@localhost
        Validity
            Not Before: Oct 20 08:13:00 2008 GMT
            Not After : Oct 20 08:13:00 2009 GMT
        Subject: C=DE, ST=NRW, O=TestCooperation, OU=IT,
        CN=cf3580f613d52db8535b7a5f6d02c3ef751f3ad7
        /emailAddress=l@test.de
        Subject Public Key Info:
            Public Key Algorithm: rsaEncryption
            RSA Public Key: (1024 bit)
                Modulus (1024 bit):
                    00:a4:f8:37:36:83:95:36:e2:e5:3b:53:64:b3:63:
                    0e:49:18:0e:3a:d5:98:d0:fb:f4:b9:74:3c:9c:b7:
                    4d:a5:b8:26:ad:a0:2d:94:b3:4f:26:ca:a2:87:08:
                    19:de:40:f8:58:48:75:66:ef:98:cf:8b:06:67:dd:
                    5b:19:db:7a:05:c6:dd:e1:f4:f3:25:38:98:69:25:
                    f0:cf:7f:bb:d9:d0:f6:97:8d:3f:d0:1c:fd:10:38:
                    f3:f9:21:b7:52:b9:ea:67:9b:5d:75:17:8a:bf:01:
                    c5:25:86:e1:f6:47:b7:3e:46:ed:00:9b:db:b0:11:
```

```
                a4:f1:f2:68:00:5c:37:97:0b
            Exponent: 65537 (0x10001)
    X509v3 extensions:
        X509v3 Basic Constraints:
            CA:FALSE
        Netscape Comment:
            OpenSSL Client Certificate
        X509v3 Subject Key Identifier:
            E4:67:63:0B:78:D5:EC:92:1C:A9:4D:23:A9:13:50:
            26:B7:CC:D3:49
        X509v3 Authority Key Identifier:
            keyid:49:2E:B7:B8:A8:89:C4:0E:AA:3D:26:BC:30:
            0E:CC:BC:EB:00:AE:00

Signature Algorithm: sha1WithRSAEncryption
    3b:64:e8:d9:66:71:6c:03:f2:a3:bb:96:62:6c:76:57:d9:65:
    6c:60:26:1f:15:92:a4:a3:a2:9f:fa:7e:39:b1:52:32:cb:0a:
    c4:c0:3b:e4:d9:e8:3a:63:f2:09:be:7d:57:71:ba:11:3a:3f:
    86:7e:c6:7b:54:36:1f:6c:fc:bb:6f:67:c9:d4:1b:a5:48:5f:
    e0:3d:e6:59:8d:0a:f4:f3:78:97:50:dc:b5:ef:68:14:6d:29:
    21:7d:3e:36:57:10:32:f7:89:66:4d:e6:32:ce:40:ad:8e:dc:
    81:ff:56:70:07:40:13:8e:c9:1f:80:83:c3:f4:ec:00:59:48:
    c1:9f:e4:33:5d:f6:85:a0:8d:f2:9f:45:a5:01:1f:b2:09:c2:
    fc:a5:a4:f5:c9:d0:56:8f:41:8c:d2:17:c3:8c:55:37:72:ed:
    cc:d1:6f:31:7d:72:c4:43:31:bf:81:9a:a9:29:07:a3:8a:38:
    0a:05:d8:a8:34:9f:0b:e5:2e:a2:83:3b:03:f4:90:b9:c9:7c:
    38:cb:af:78:3c:d2:9d:fc:b5:13:ed:ea:95:2d:19:8d:41:5c:
    6a:40:ff:6b:ec:44:05:59:c1:22:36:dc:e2:0c:3f:62:d7:6f:
    a9:3f:d1:4f:c8:6c:75:5f:ec:e5:98:5a:2e:79:e3:13:7a:f6:
    8a:56:fd:68
```

To calculate the size of the certificate the base 64 representation is used. No leading blanks and other fill characters appear after the conversion. The used text is given below.


```
-----BEGIN CERTIFICATE-----
MIIDfTCCAmWgAwIBAgIBFjANBgkqhkiG9w0BAQUFADB+MQswCQYDVQQGEwJERTEM
MAoGA1UECBMDTlJXMRkwFwYDVQQKExBUZXN0IENvb3BlcmF0aW9uMRMwEQYDVQQL
EwpJVCBzZWN0aW9uMRAwDgYDVQQDEwdVc2VyIENBMR8wHQYJKoZIhvcNAQkBFhB1
c2VyQ0FAbG9jYWxob3N0MB4XDTA4MTAyMDA4MTMwMFoXDTA5MTAyMDA4MTMwMFow
gY8xCzAJBgNVBAYTAkRFMQwwCgYDVQQIEwNOUlcxGDAWBgNVBAoTD1Rlc3RDb29w
ZXJhdGlvbjELMAkGA1UECxMCSVQxMTAvBgNVBAMTKGNmMzU4MGY2MTNkNTJkYjg1
MzViN2E1ZjZkMDJjM2VmNzUxZjNhZDcxGDAWBgkqhkiG9w0BCQEWCWxAdGVzdC5k
ZTCBnzANBgkqhkiG9w0BAQEFAAOBjQAwgYkCgYEApPg3NoOVNuLlO1Nks2MOSRgO
OtWYOPvOuXQ8nLdNpbgmraAtlLNPJsqihwgZ3kD4WEh1Zu+Yz4sGZ91bGdt6Bcbd
4fTzJTiYaSXwz3+72dD2l40/0Bz9EDjz+SG3UrnqZ5tddReKvwHFJYbh9ke3Pkbt
```

```
AJvbsBGk8fJoAFw3lwsCAwEAAaN4MHYwCQYDVR0TBAIwADApBglghkgBhvhCAQ0E
HBYaT3BlblNTTCBDbGllbnQgQ2VydGlmaWNhdGUwHQYDVR0OBBYEFORnYwt41eyS
HKlNI6kTUCa3zNNJMB8GA1UdIwQYMBaAFEkut7ioicQOqj0mvDAOzLzrAK4AMA0G
CSqGSIb3DQEBBQUAA4IBAQA7ZOjZZnFsA/Kju5ZibHZX2WVsYCYfFZKko6Kf+n45
sVIyywrEwDvk2eg6Y/IJvn1XcboROj+GfsZ7VDYfbPy7b2fJ1BulSF/gPeZZjQr0
83iXUNy172gUbSkhfT42VxAy94lmTeYyzkCtjtyB/1ZwB0ATjskfgIPD9OwAWUjB
n+QzXfaFoI3yn0WlAR+yCcL8paT1ydBWj0GM0hfDjFU3cu3M0W8xfXLEQzG/gZqp
KQejijgKBdioNJ8L5S6igzsD9JC5yXw4y694PNKd/LUT7eqVLRmNQVxqQP9r7EQF
WcEiNtziDD9i12+pP9FPyGx1X+zlmFoueeMTevaKVv1o
-----END CERTIFICATE-----
```

The size of this certificate is 1269 bytes long. An advantage of a standard certificate is the long-life cycle. Once transferred, a client can save it for future use. For a signature of a text, not only the certificate must be transferred but also the signed message. In this case the algorithm used is 1024 bit RSA, so the signature is 128 bytes long. Therefore the final size is 1397 bytes.

B.2 Dynamic Cryptographic Accumulator Signature

In the signature protocol, see Section 5.6, of this dynamic cryptographic accumulator scheme certain values are transferred. In combination, these values assemble the signature. A certificate as presented in the section above is not necessary. The values are:

$$(E, \Lambda, U_1, U_2, R, T_1, T_2, T_3, \Pi_1, \Pi_2, \Pi_3, s_0, s_1, s_2, s_3, s_4, s_5, s_6, s_7, s_8)$$

The size of the individual elements depends on the chosen pairing. All possible sizes are explained in [29] on page 75.
The type A pairing is a symmetric pairing with an embedding degree of 2. It is very fast but the group size itself is relatively large. In this special case the group size is not the only relevant criterion.
The pairing type F is an asymmetric pairing with an embedding degree of 12. The group $\mathbb{G}_1$ is very small but the group $\mathbb{G}_2$ is, due to the degree, very large. One of the main drawbacks of all asymmetric pairings is the speed. Due to the complexity of the pairing the operations are much slower.
The different pairings result in different signature sizes which are described below.

The complete size of the signatures is now calculated.

Elements	Type A	Type F
E	513	161
Λ	1024	1920
U_1	513	161
U_2	513	161
R	513	161
T_1	513	161
T_2	513	161
T_3	513	161
Π_1	1024	1920
Π_2	1024	1920
Π_3	1024	1920
s_0	160	160
s_1	160	160
s_2	160	160
s_3	160	160
s_4	160	160
s_5	160	160
s_6	160	160
s_7	160	160
s_8	160	160

Table B.1: Comparison of Pairings

$$\begin{aligned}
Type\ A: \quad & 7 * 513 = 3591 \\
& 4 * 1024 = 4096 \\
& 9 * 160 = 1440 \\
& \rightarrow 9127\ bits \\
& \rightarrow 1141\ bytes \\
Type\ F: \quad & 7 * 161 = 1127 \\
& 4 * 1920 = 7680 \\
& 9 * 160 = 1440 \\
& \rightarrow 10247\ bits \\
& \rightarrow 1281\ bytes
\end{aligned}$$

As a result, the type A pairing is selected for the scheme. Maybe newer and better suited pairings may be presented in the near future which can replace this choice. The resulting signature is smaller than the standard signature combined with a certificate. When only the signature without the certificate is sent to a verifier the standard signature is shorter.

C Speed Comparison with PKI-based Certificates

Here our implemented group signature scheme is compared to the OpenSSL applications. Both projects have different security requirements and foci in the utilization. But the OpenSSL project is one of the most popular signature implementations today. So it is one of the best candidates to compare to.
The times below are the arithmetic average of ten runs of the program. The applications were compiled by the GNU compiler version 4.3.2-1ubuntu11 and ran on a Intel Core 2 Duo E6750 $2 \times 2.66\,GHz$ under Linux Ubuntu 8.10.

Scheme	GROUP GENERATION	JOIN STAGE	SIGN STAGE	VERIFY STAGE
OpenSSL	$0.151\,s$	$0.100\,s$	$0.009\,s$	$0.007\,s$
GSS	$0.189\,s$	$0.900\,s$	$0.708\,s$	$0.800\,s$

Table C.1: Speeds of different Operations

In the group generation phase the OpenSSL application creates a random file and a root certificate. In the case of the accumulator scheme the group key generation algorithm is executed.
Notice that the join stage is just the creation of a certificate when using OpenSSL. Our implementation uses a network protocol and depends heavily on the network speed, so the stated times can vary in practice.
The large differences in the sign and verify stage result mostly from the mathematical equations used in the implementations. In case of OpenSSL, a signature is created and verified with only one exponentiation. The group siganture verify algrithm needs 22 multiplications, 7 additions, 15 exponentiations and 13 pairing operations.