Eidgenössische Technische Hochschule
Swiss Federal Institute of Technology
Zürich

Department of Computer Science
Institute of Theoretical Computer Science

SMS Lottery

An Application for MOVA

Florin Oswald

Master's Thesis in Computer Science

March 20th, 2006 – September 19th, 2006

Advisor: Prof. Ueli Maurer
Information Security and Cryptography Research Group
Swiss Federal Institute of Technology Zurich

Supervisor: Prof. Serge Vaudenay
Security and Cryptography Laboratory (LASEC)
Swiss Federal Institute of Technology Lausanne

This thesis is submitted for partial fulfillment of the requirements of the degree Master of Science in Computer Science at the Swiss Federal Institute of Technology (ETH) Zurich. The thesis was derived during a six month project from March 20th *2006 to September* 19th *2006 in the Security and Cryptography Laboratory (LASEC) led by Prof. Serge Vaudenay at the Department of Computer Science of the Swiss Federal Institute of Technology Lausanne.*

Acknowledgements

First of all, I would like to thank Professor Serge Vaudenay for welcoming me in the LASEC laboratory and for supervising my Master's Thesis. I highly appreciated that he was always willing to spend some of his valuable time for proofreading and discussing technical topics. Also, I would like to thank Professor Ueli Maurer for accepting to be the responsible professor at the ETH for my Master's Thesis. He facilitated the transfer to the EPFL in a very unbureaucratic way. Special thanks go also to Jean Monnerat, for helping me to understand the MOVA signature scheme and for the stimulating discussions, which often took place late in the evenings and during the weekends. Furthermore I would like to thank Sylvain Pasini for providing me the source code of his MOVA application, which was a very good starting point for my own application.

This work is dedicated to my family and friends who have given me strong support during my studies at ETH.

Lausanne, September 19th, 2006.

Florin Oswald

Abstract

With SMS Lottery we propose an application for the undeniable signature scheme MOVA. In contrast to traditional SMS based lottery systems, our system offers cryptographic security to prove that the lottery organization has registered of a SMS lottery ticket. We analyze different solutions and propose a system that is in our opinion optimal to be implemented in practice. To prove the security of our system, we define a security model and deduce the required security parameters from it. A Java implementation of the system proves the feasibility of the concept.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Digital signatures are widely used in public key infrastructures to assure the authenticity of digital objects. In 1989, David Chaum and Hans van Antwerpen [CA89] have introduced the concept of *undeniable signatures*, which are in contrast to traditional digital signatures not universally verifiable. The verification of the signature is done by the execution of an interactive protocol, where the signer convinces the verifier of the (in)validity of the signature. Numerous undeniable signature schemes have been proposed since then. In 2004, Serge Vaudenay and Jean Monnerat have developed the undeniable signature scheme MOVA, that is based on group homomorphisms and allows particularly short signatures. Even if it was quite revolutionary to have such short signatures, it was not clear whether there are applications where this scheme can generate added value in comparison to traditional signature schemes.

Since the appearance of the SMS technology for mobile phones, more and more providers make their services available per SMS. The query of timetables and the order of ring tones are two very popular examples. The possibility to access a service at any possible place generates customer's benefits and is therefore an important opportunity for the providers. This has also been understood by providers of gambling games: Nowadays it is very easy to bet some money on your favorite football club by just sending an SMS. In some countries it is also possible to play lottery per SMS. These games where real money is involved raise new security issues, such as non-repudiation of the player's stake.

In this work, we examine the application of SMS lottery. We show that the use of short undeniable signatures allows the players to have more confidence in the system. After examining different settings, we choose a protocol that is in our opinion the most adequate to be implemented in practice. For this protocol, we make a rigorous security analysis. To show its practicability, we have implemented the protocol in a Java tool called the *SMS Lottery Simulator*.

In Section 2 we review some fundamental cryptographic notions. Section 3 contains a short overview of the MOVA signature scheme including the most important definitions, theorems and protocols. In Section 4 we develop the SMS lottery application. The *SMS Lottery Simulator* is described in Section 5.

Chapter 2

Preliminaries

2.1 Basic Cryptographic Notions

In this section we review some basic cryptographic notions in an informal way.

2.1.1 Success Probability and Advantage

For a given problem $\mathcal{P}$, we denote by $\mathsf{Succ}_{\mathcal{A}}^{\mathcal{P}}$ the success probability of an attacker $\mathcal{A}$ to solve the problem $\mathcal{P}$.

If a problem is *decisional*, we consider a *distinguisher* rather than an *attacker*. Instead of the *sucess probability* we rather consider the *advantage*. We denote by $\mathsf{Adv}_{\mathcal{D}}^{\mathcal{P}}$ the advantage for a distinguisher $\mathcal{D}$ to solve the decisional problem $\mathcal{P}$. The advantage is defined as follows:

$$\mathsf{Adv}_{\mathcal{D}}^{\mathcal{P}} := |\Pr[0 \leftarrow \mathcal{D}(x) \mid x \leftarrow D_0] - \Pr[0 \leftarrow \mathcal{D}(x) \mid x \leftarrow D_1]|$$

where D_0 and D_1 are two different distributions and the distinguisher $\mathcal{D}$ has to decide whether x was drawn according to the distribution D_0 or D_1.

2.1.2 One-Way Function

A *one-way function* is an efficiently calculable function f from a set A to a set B, $f : A \rightarrow B$, such that it is computationally difficult to find for a given $b \in B$ an $a \in A$ such that $f(a) = b$.

A *trapdoor one-way function* is an efficiently calculable function f from a set A to a set B and a secret $\mathcal{K}_{\mathrm{s}}$, such that without knowing $\mathcal{K}_{\mathrm{s}}$ it is difficult to find for a given $b \in B$ an $a \in A$ such that $f(a) = b$. *With* knowledge of $\mathcal{K}_{\mathrm{s}}$, this can be done efficiently.

2.1.3 Hash Function

A *hash function* is an efficiently calculable function H from a set A to a set B, $H : A \to B$, such that following conditions are satisfied:

- *Preimage Resistance*: Given $b \in B$ it is computationally hard to find any a such that $H(a) = b$.
- *Second Preimage Resistance*: Given $a_1 \in A$ it is computationally hard to find another value $a_2 \in A$ (not equal to a_1) such that $H(a_1) = H(a_2)$.
- *Collision Resistance*: It is computationally hard to find two different values $a_1, a_2 \in A$ such that $H(a_1) = H(a_2)$.

2.1.4 Pseudorandom Generator

A *pseudorandom generator* is an algorithm that takes as input a *seed* that is picked uniformly at random, and generates a sequence of numbers such that there is no efficient *distinguisher* that distinguishes this sequence from a sequence of true random numbers.

2.1.5 Message Authentication Codes

A *Message Authentication Code* (MAC) is a piece of information used to authenticate a digital document. A MAC algorithm takes a digital document and a secret key, and generates a MAC. Anybody who possesses the secret key can be assured, that this MAC was generated by someone who also possess the secret key. Therefore we have the propriety of *unforgeability*: Nobody who does not possess the secret key, is able to generate a valid MAC to a message.

2.1.6 Digital Signatures

Definition 1. *A digital signature scheme consists of:*

- *A probabilistic key generation algorithm* $(\mathcal{K}_\mathrm{s}, \mathcal{K}_\mathrm{p}) \leftarrow \mathsf{KGen}(1^k)$ *for a security parameter* k*, which generates a public key* $\mathcal{K}_\mathrm{p}$ *and a secret key* $\mathcal{K}_\mathrm{s}$*.*
- *A signature algorithm* $sig \leftarrow \mathsf{Sign}(m, \mathcal{K}_\mathrm{s})$*, which computes (in a probabilistic or deterministic way) a signature* sig *from a message* m *and the secret key* $\mathcal{K}_\mathrm{s}$*.*
- *A verification algorithm* $v \leftarrow \mathsf{Verify}(m, sig, \mathcal{K}_\mathrm{p})$*, which from a message* m*, signature* sig*, and public key* $\mathcal{K}_\mathrm{p}$ *verifies (in a deterministic way) the correctness of the signature and outputs* $v \in \{0, 1\}$ *corresponding to the result.*

Such a digital signature scheme must satisfy following security properties:

- *Authenticity and integrity (*unforgeability*): It must be impossible for anyone who does not have access to the secret key to forge a* (m, sig) *pair which is valid for the public key* $\mathcal{K}_\mathrm{p}$*.*

- *Non-repudiation: It must be impossible for the legitimate signer to repudiate his signature. When a signed message (m, sig) is valid, the signer cannot claim that the signature was forged. The property of* non-repudiation *is implied by the property of* unforgeability.

Note that in contrast to *undeniable signature schemes*, *digital signature schemes* imply the property of *universal verifiability.* This means that the validity of a signature can be verified locally, without interaction with the signer.

The *unforgeability* property we have stated in Definition 1 corresponds to the security against an *existential forgery.* We can also consider weaker definitions of unforgeability:

Definition 2. *A signature scheme is* secure against total break*, if there does not exist any polynomially limited adversary that wins Game^{tb} with a non-negligible probability. Game^{tb} is defined as follows: The adversary knows only the* public key *and wins the game, if she outputs the corresponding* secret key.

Definition 3. *A signature scheme is* secure against universal forgery*, if there does not exist any polynomially limited adversary that wins Game^{uf} with a non-negligible probability. Game^{uf} is defined as follows: The adversary knows only the* public key *and receives a message m that is chosen uniformly at random from the message space. She has to output a signature for m and wins, if the signature is valid.*

Definition 4. *A signature scheme is* secure against existential forgery*, if there does not exist any polynomially limited adversary that wins Game^{ef} with a non-negligible probability. Game^{ef} is defined as follows: The adversary knows only the* public key*. She wins the game, if she outputs a pair (m, sig) where sig is a valid signature for the message m.*

We denote by Game^{uf-cma} the universal forgery game where the adversary has additionally access to a Sign oracle, which computes the valid signature for a message m that is chosen by the adversary. She can query any message to this oracle beside the message for which she needs to forge the signature.

We denote by Game^{ef-cma} the existential forgery game where the adversary has additionally access to a Sign oracle. Obviously she only wins the game if she outputs a message m together with a valid signature sig, for which she has not previously queried the Sign oracle.

2.1.7 Interactive Proofs

In an interactive proof, a *Prover* tries to convince a *Verifier* of some assertion by exchanging messages with him. The basic requirements to an interactive proof protocol include:

- *Completeness*: If the assertion is true, the verifier accepts the proof.

- *Soundness*: If the assertion is false, there is no strategy for the prover to convince the verifier.

Additionally following properties for interactive proof protocols are relevant for this work:

- *Zero-Knowledge*: An execution of the protocol does not reveal any other information to the verifier but the fact that the assertion is true.

- *Non-Transferability*: A verifier is not able to transfer the proof to a third-party, e.g. she can not convince any other party of the validity of the assertion.

We will use following different definitions of *zero-knowledge* in this work:

Definition 5. *An interactive proof protocol satisfies **perfect zero-knowledge** if there exists for every verifier a* simulator *who transforms the verifier into an algorithm which outputs protocol transcripts which have the same probability distribution as the protocols generated by protocol executions between a honest prover and the verifier.*

Definition 6. *If there does not exist any computationally unbounded distinguisher that is able to distinguish the two transcripts with a non-negligible probability, we call an interactive proof protocol **statistically zero-knowledge**.*

Definition 7. *If the simulator is only allowed to use the verifier as a* black-box*, we denote it as **perfect black-box zero-knowledge** or **statistically black-box zero-knowledge** respective.*

Definition 8. *If the simulator is not allowed to rewind the verifier, we denote it as **perfect black-box straight-line zero-knowledge** or **statistically black-box straight-line zero-knowledge** respective.*

2.1.8 Undeniable Signatures

In a conventional digital signature scheme, the verification of a signature can be done without communicating with the signer, the signer's public key is sufficient. The notion of *Undeniable Signature Scheme* was introduced by David Chaum and Hans van Antwerpen [CA89]. It denotes a scheme where the verification of a signature can only be done by executing an interactive protocol between the signer and the verifier.

An undeniable signature scheme consists of following parts:

- A key probabilistic generation algorithm $(\mathcal{K}_\mathrm{p}, \mathcal{K}_\mathrm{s}) \leftarrow \mathsf{GenK}(1^k)$ that takes a security parameter k and outputs a public key $\mathcal{K}_\mathrm{p}$ and a secret key $\mathcal{K}_\mathrm{s}$.

- A signature algorithm, $sig \leftarrow \mathsf{Sign}(m, \mathcal{K}_\mathrm{s})$ which takes a message m and the secret key $\mathcal{K}_\mathrm{s}$ and produces a signature sig.

- A confirmation protocol *Confirmation*, in which the signer can prove to the verifier that a given signature is valid, e.g. could only have been generated with knowledge of the secret key $\mathcal{K}_{\mathrm{s}}$ which belongs to the public key $\mathcal{K}_{\mathrm{p}}$.
- A denial protocol *Denial*, in which the signer can prove to the verifier that a given signature is not valid, e.g. it was not generated with the secret key which belongs to the public key.

Such a scheme ideally satisfies following security requirements:

- *Unforgeability*: Nobody can generate a valid signature to a message without knowing the secret key.
- *Soundness for confirmation*: If a signature is not valid, nobody can successfully convince the verifier that it is valid.
- *Soundness for denial*: If a signature is valid, nobody can successfully convince the verifier that it is not valid.
- *Zero-knowledge*: The confirmation and denial protocols do not reveal any other information than the (in-)validity of the signature.
- *Invisibility*: Nobody can distinguish a valid signature from an invalid one without knowing the secret key.
- *Non-transferability*: Nobody can convince another person that a signature is valid without knowing the secret key.

For the *unforgeability* property we consider the same classes as specified in the Section 2.1.6 (total break, universal forgery, existential forgery).

Note that the name *undeniable* is misleading. Obviously every signature scheme should be *undeniable*, not only *undeniable signature schemes*, but also conventional signature schemes. A more adequate name would be *interactively verifiable signature schemes.*

2.1.9 Commitment Schemes

Definition 9. *Formally, a commitment scheme consists of two algorithms:*

- *An algorithm* $(\mathsf{com}, \mathsf{dec}) \leftarrow \mathsf{Commit}(m)$ *which takes as input a message* m *and outputs a* commitment com *and the opening information* dec.
- *An algorithm* $v \leftarrow \mathsf{Open}(m, \mathsf{com}, \mathsf{dec})$ *that checks whether the commited value corresponds to the message* m *and the opening information* dec *and outputs* $v \in \{0, 1\}$ *depending on the result.*

A commitment scheme is *perfectly binding* if the commitment com completely defines the message m. It is *computationally binding*, if it is computationally difficult to find two different messages $m \neq m'$, a commitment com and two opening informations dec and dec' such that both $\mathsf{Open}(m, \mathsf{com}, \mathsf{dec})$ and $\mathsf{Open}(m', \mathsf{com}, \mathsf{dec}')$ accept.

More formally, we can define the *computationally binding* property on behalf of a game:

Definition 10. *A commitment scheme is* computationally binding*, if there does not exist any polynomially adversary $\mathcal{A}$ that wins the following game with non-negligible probability:*
$\mathsf{Game}^{com-bnd}$: *$\mathcal{A}$ wins the game if she outputs some elements* $(\mathsf{com}, m, \mathsf{dec}, m', \mathsf{dec}')$ *such that* Open *accepts* $(m, \mathsf{com}, \mathsf{dec})$ *and* $(m', \mathsf{com}, \mathsf{dec}')$ *and $m \neq m'$.*

A commitment scheme is *perfectly hiding*, if for any message m, the distribution of com is the uniform distribution. It is *computationally hiding*, if there is no pair (m, m') with $m \neq m'$ such that there exists an efficient algorithm that can determine if a commitment com is for m or for m'.

In a *trapdoor commitment scheme*, it is easy to find a collision for someone that possesses the secret key. More formally, a *trapdoor commitment scheme* consists of four algorithms:

- A probabilistic algorithm that generates a key pair $(\mathcal{K}_{\mathrm{s}}^{\mathbf{V}}, \mathcal{K}_{\mathrm{p}}^{\mathbf{V}}) \leftarrow \mathsf{Setup}^{\mathbf{V}}(1^k)$ where k is the security parameter.

- An algorithm $(\mathsf{com}, \mathsf{dec}) \leftarrow \mathsf{Commit}(\mathcal{K}_{\mathrm{p}}^{\mathbf{V}}, m)$ that takes a message m and the public key $\mathcal{K}_{\mathrm{p}}^{\mathbf{V}}$ and outputs two values com and dec.

- An algorithm $v \leftarrow \mathsf{Open}(\mathcal{K}_{\mathrm{p}}^{\mathbf{V}}, m, \mathsf{com}, \mathsf{dec})$ that checks whether a commited value com corresponds to a message m and the opening information dec and outputs $v \in \{0, 1\}$ depending on the result.

- An algorithm $\mathsf{dec}' \leftarrow \mathsf{Collide}(\mathcal{K}_{\mathrm{s}}^{\mathbf{V}}, m, m', \mathsf{com}, \mathsf{dec})$ that receives two messages $m \neq m'$ and values dec and com, such that Open accepts $(m, \mathsf{dec}, \mathsf{com})$, and the secret key $\mathcal{K}_{\mathrm{s}}^{\mathbf{V}}$. It then outputs a value dec', such that Open also accepts $(m', \mathsf{com}, \mathsf{dec}')$.

We say that a *trapdoor commitment scheme* is *computationally binding* if there exists no probabilistic polynomial time algorithm $\mathcal{A}$ which wins the following game with a non-negligible probability.

2.1.10 Random Oracles

A *Random Oracle* is a theoretical concept, which implements a function RO from a set A to a set B. For every query $a \in A$, it responds with a (truly) random b which is chosen uniformly from B. Every time it receives the same query, it replies with the same answer. Some cryptographic proofs require the (theoretical) existence of random oracles. A system that is proven secure assuming the existence of random oracles, is called *secure in the random oracle model*.

Note that the security proofs of this work are done in the random oracle model!

Chapter 3

MOVA Signature Scheme

The MOVA signature scheme is an undeniable signature scheme that was first proposed at Asiacrypt 2004 by Jean Monnerat and Serge Vaudenay [MV04]. It allows particularly small sizes of signatures which makes it optimal for applications where the signature has to be stored on a small memory or where it needs to be transmitted over a medium with low data bandwidth (e.g. human voice).

This section is organized as follows: In Section 3.1 we introduce some notions and auxiliary protocols that will be used in Section 3.2, where we introduce the MOVA signature scheme.

The definitions, theorems and protocols in this chapter are taken from [M06].

3.1 Preliminaries

3.1.1 Group Homomorphism Interpolation

The secret key of a MOVA signature scheme is a group homomorphism[1] $\mathrm{Hom} : G \rightarrow H$ from a group G to a group H. Informally, we can say that a message defines some elements in G, and the corresponding signature consists of the respective elements in H defined by the group homomorphism.

The public key consists of pairs (g_i, h_i) for $g_i \in G$ and $h_i \in H$ such that $\mathrm{Hom}(g_i) = h_i$ for $i = 1, ..., \mathrm{Lkey}$, where Lkey is the length of the key. The number of these pairs has to be chosen such that it is very improbable that there exist more than *one* homomorphism which interpolates these pairs. We will define more below what *improbable* means here.

If the signer wants to prove the validity of a signature sig for a message msg, she needs to show that for the $x_j \in G$ which are defined by msg and for the $y_i \in H$ which are defined by sig, $\mathrm{Hom}(x_j) = y_j$ is satisfied for $j = 1, ..., \mathrm{Lsig}$, where Lsig is the length of the signature, and Hom is the homomorphism that is defined by the public key. We say, that she has to show that the pairs (x_j, y_j) have to interpolate with the (g_i, h_i) in a group homomorphism. Similarly, to prove the invalidity of a signature, she has to show that the elements of H that correspond

[1]To be more precise, we have to say that the secret key consists of the information needed to efficiently calculate a given group homomorphism.

to the signature, do not interpolate with g_i in a group homomorphism. The exact definition is given as follows:

Definition 11. *Let G, H be two Abelian groups and S a subset of $G \times H$ written in the form $S := \{(x_1, y_1), ..., (x_s, y_s)\}$.*

1. *We say that the set of points S* interpolates in a group homomorphism *if there exists a group homomorphism $f : G \longrightarrow H$ such that $f(x_i) = y_i$ for $i = 1, ..., s$.*

2. *We say that a set of points $B \subseteq G \times H$* interpolates in a group homomorphism with another set of points *$A \subseteq G \times H$ if $A \cup B$ interpolates in a group homomorphism.*

3.1.2 Group Homomorphism Interpolation Problem

In Section 4.4.3 we will use the Group Homomorphism Interpolation Problem to prove the security of the SMS Lottery system by means of a security reduction. The Group Homomorphism Interpolation Problem is defined as follows:

Definition 12. (**n-S-GHI Problem**, n-S-Group Homomorphism Interpolation Problem)

Parameters: *Two Abelian groups G and H, a set $S \subseteq G \times H$ of s points, and a positive integer n.*

Instance Generation: *n elements $x_1, \ldots, x_n$ picked uniformly at random in G.*

Problem: *Find $y_1, \ldots, y_n \in H$ such that $\{(x_1, y_1), \ldots, (x_n, y_n)\}$ interpolates with S in a group homomorphism.*

3.1.3 Group Homomorphism Interpolation Decisional Problem

The *decisional* version of the Group Homomorphism Interpolation Problem is defined as follows:

Definition 13. (**n-S-GHID Problem**, n-S-GHI Decisional Problem)

Parameters: *Two Abelian groups G and H, a set $S \subseteq G \times H$ of s points, and a positive integer n.*

Instance Generation: *The instance T is generated according to one of the two following ways and is denoted T_0 or T_1 respectively. T_0 is a set of points $\{(x_1, y_1), \ldots, (x_n, y_n)\} \in (G \times H)^n$ picked uniformly at random such that it interpolates with S in a group homomorphism. T_1 is picked uniformly at random in $(G \times H)^n$.*

Problem: *Decide whether the instance T is of type T_0 or T_1.*

3.1.4 H-generation of G

For the security of the MOVA signature scheme it is important that the pairs (g_i, h_i) from the public key define *at most one* homomorphism from G to H. Otherwise, the signer could know two different homomorphisms (which means that she knows two different secret keys for a given public key). We denote this property as g_i H-generate G. More formally:

Definition 14. *Let G, H be two finite Abelian groups. Let $x_1, \ldots, x_s \in G$ which span a subgroup denoted by G'. We say that $x_1, \ldots, x_s$ H-generate G if for any elements $y_1, \ldots, y_s \in H$, there exists at most one group homomorphism $f : G \longrightarrow H$ such that $f(x_i) = y_i$ for all $i = 1, \ldots, s$.*

3.1.5 Expert Group Knowledge

Expert Group Knowledge is an ability that can be achieved by the prover in some settings, e.g. if the groups G and H and the homomorphism is chosen accordingly. It is defined as follows:

Definition 15. *Let G, H be two finite Abelian groups. We denote d the order of H. Let $S = x_1, \ldots, x_s \in G$, such that they H-generate G. We say that one has an* Expert Group Knowledge *of G with the set S, if one is able to find for every $x \in G$ some coefficients $a_1, \ldots, a_s \in \mathbb{Z}_d$ and $r \in G$, such that $x = dr + a_1x_1 + \ldots + a_sx_s$.*

3.1.6 Proof Protocol for the GHID Problem

With the following protocol, a prover can convince a verifier that a set $S = \{(g_1, e_1), ..., (g_s, e_s)\}$ interpolates in a group homomorphism $f : G \rightarrow H$. We will use this protocol later as a sub-protocol in the confirmation protocol of the MOVA scheme, where the signer can prove that a signature is valid. The security parameters are $d := |H|$ and $\ell \in \mathbb{N}$.

GHIproof$_\ell(S)$:
Parameters: G, H, d
Input: ℓ, $S = \{(g_1, e_1), \ldots, (g_s, e_s)\} \subseteq G \times H$

```
1: The verifier picks $r_i \in_U G$ and $a_{i,j} \in_U \mathbb{Z}_d$ uniformly at random for $i = 1, \ldots, \ell$ and $j = 1, \ldots, s$. He computes $u_i = dr_i + a_{i,1}g_1 + \cdots + a_{i,s}g_s$ and $w_i = a_{i,1}e_1 + \cdots + a_{i,s}e_s$ for $i = 1, \ldots, \ell$. He sends $u_1, \ldots, u_\ell$ to the prover.
2: The prover checks whether $f(g_i) = e_i$ for $i = 1, \ldots, \ell$. If it is not the case, he aborts the protocol. Then, he computes $v_i = f(u_i)$ for $i = 1, \ldots, \ell$ and the commitment $(\mathsf{com}, \mathsf{dec}) \leftarrow \mathsf{Commit}(v_1, \ldots, v_\ell)$. He sends the committed value com to the verifier.
3: The verifier sends all $r_i$'s and $a_{i,j}$'s to the prover.
4: The prover checks that the $u_i$'s were computed correctly by verifying that $u_i = dr_i + a_{i,1}g_1 + \cdots + a_{i,s}g_s$ holds for $i = 1, \ldots, \ell$. If not, he aborts the protocol. He then opens his commitment by sending the values $v_i$'s and dec.
```

5: The verifier checks that $v_i = w_i$ holds for $i = 1, \ldots, \ell$ and that the commitment is opened correctly, i.e., $1 \leftarrow \mathsf{Open}(v_1, \ldots, v_\ell, \mathsf{com}, \mathsf{dec})$. If this is the case, the verifier accepts the proof. Otherwise, he rejects it.

3.1.7 Properties of Proof Protocol for the GHID Problem

This proof protocol for the GHID problem satisfies following security properties (a proof can be found in [M06]):

Theorem 1. *Let G, H be some Abelian groups and $\ell \in \mathbb{N}$. We denote by d the order of H and p the smallest prime factor of d. Assume that we are given a set of points $S = \{(g_1, e_1), \ldots, (g_s, e_s)\} \subseteq G \times H$ such that the elements $g_1, \ldots, g_s$ H-generate the group G. We consider the* GHIproof$_\ell(S)$ *protocol.*

1. *The* GHIproof *protocol is complete.*
2. *Assuming that* Commit *is perfectly hiding, the* GHIproof *protocol is perfect black-box zero-knowledge against any verifier. Moreover, if* Commit *is a perfectly hiding trapdoor commitment scheme, the* GHIproof *protocol is perfect black-box straight-line zero-knowledge against any verifier who has the secret key $\mathcal{K}_s^{\mathbf{V}}$ associated to* Commit.
3. *Assuming that* Commit *is a perfectly hiding trapdoor commitment scheme with associated secret key $\mathcal{K}_s^{\mathbf{V}}$ of the (honest) verifier, the* GHIproof *protocol is perfect non-transferable.*
4. *The* GHIproof *protocol is sound: from any cheating prover $\mathbf{P}^*$ who passes the protocol on an invalid set S (i.e. with no interpolating homomorphism) with a probability $\mathsf{Succ}_{\mathbf{P}^*}^{\mathsf{sd}\text{-GHI}} = \varepsilon$ and an expert group knowledge of G, we can construct an algorithm $\mathcal{B}$ which finds a collision on the commitment scheme* Commit *with a probability $\mathsf{Succ}_{\mathcal{B}}^{\mathsf{com\text{-}bnd}} \geq \varepsilon(\varepsilon - p^{-\ell})$ by rewinding $\mathbf{P}^*$ once.*
5. *("Proof of knowledge") For any $\theta > 0$, assuming that the protocol succeeds with probability greater than $(p^{-1} + \theta)^\ell$ with an honest verifier and that* Commit *is extractable, for any $\varepsilon > 0$ there exists an extractor with a time complexity factor $\mathcal{O}(\log(1/\varepsilon))$ which can compute an interpolating group homomorphism from the (possibly cheating) prover using the secret key of* Commit *with probability at least $1 - \varepsilon$.*

3.1.8 Proof Protocol for the co-GHID Problem

With the proof protocol for the co-GHID problem, a prover can show that a set S *does not* interpolate in a group homomorphism $f : G \to H$. It will be used as a sub-protocol in the denial protocol, where the signer can prove that a signature is not valid.

Let G, H and $S = \{(g_1, e_1), ..., (g_s, e_s)\} \subseteq G \times H$ be parameters of a GHID problem, and let d be the order of H with smallest prime factor p. Let $T = \{(x_1, z_1), ..., (x_t, z_t)\} \subseteq G \times H$ be a set of t points. With the following protocol, a prover can prove that at least for one k, the pair (x_k, z_k) is not interpolating with S in a group homomorphism. The security parameter is $\ell \in \mathbb{N}$.

```
coGHIproof$_\ell(S,T)$
Parameters: $G, H, d, p$
Input: $\ell$, $S = \{(g_1,e_1),\dots,(g_s,e_s)\}$, $T = \{(x_1,z_1),\dots,(x_t,z_t)\}$
1: The verifier picks $r_{i,k} \in_U G$, $a_{i,j,k} \in_U \mathbb{Z}_d$, and $\lambda_i \in_U \mathbb{Z}_p$ uniformly at random for
   $i = 1,\dots,\ell$, $j = 1,\dots,s$, $k = 1,\dots,t$. He computes $u_{i,k} := dr_{i,k} + \sum_{j=1}^s a_{i,j,k}g_j + \lambda_i x_k$ and $w_{i,k} := \sum_{j=1}^s a_{i,j,k}e_j + \lambda_i z_k$ for all $i$ and $k$. Set $u := (u_{1,1},\dots,u_{\ell,t})$ and $w := (w_{1,1},\dots,w_{\ell,t})$. He sends $u$ and $w$ to the prover.
2: The prover computes $y_k = f(x_k)$ for $k = 1,\dots,t$ and verifies that $y_k \neq z_k$ for at least one $k$. Otherwise, he aborts the protocol. Then, he computes $v_{i,k} := f(u_{i,k})$ for $i = 1,\dots,\ell$, $k = 1,\dots,t$. From the equation $w_{i,k} - v_{i,k} = \lambda_i(z_k - y_k)$, he should be able to find each $\lambda_i$ if the verifier is honest since $y_k \neq z_k$ for at least one $k$. Otherwise, he picks $\lambda_i \in_U \mathbb{Z}_p$ uniformly at random for $i = 1,\dots,\ell$. He computes $(\mathsf{com},\mathsf{dec}) \leftarrow \mathsf{Commit}(\lambda)$, where $\lambda := (\lambda_1,\dots,\lambda_\ell)$. The prover sends the committed value com to the verifier.
3: The verifier sends all $r_{i,k}$'s and $a_{i,j,k}$'s to the prover.
4: The prover checks that $u$ and $w$ were correctly computed by verifying that $u_{i,k} := dr_{i,k} + \sum_{j=1}^s a_{i,j,k}g_j + \lambda_i x_k$ and $w_{i,k} := \sum_{j=1}^s a_{i,j,k}e_j + \lambda_i z_k$ for all $i$ and $k$. If not, he aborts the protocol. He then opens the commitment by sending $\lambda$ and dec.
5: The verifier checks that the prover has found the right $\lambda$ and that the commitment is correctly opened by checking $1 \leftarrow \mathsf{Open}(\lambda,\mathsf{com},\mathsf{dec})$. If this is the case, the verifier accepts the proof. Otherwise, he rejects it.
```

3.1.9 Properties of Proof Protocol for the co-GHID Problem

This proof protocol for the co-GHID problem satisfies following security properties (a proof can be found in [M06]):

Theorem 2. *Let G, H be some Abelian groups and $\ell \in \mathbb{N}$. We denote by d the order of H and p the smallest prime factor of d. Assume that we are given a set of points $S = \{(g_1,e_1),\dots,(g_s,e_s)\} \subseteq G \times H$ such that the elements $g_1,\dots,g_s$ H-generate the group G and such that S interpolates in exactly one homomorphism f known by the prover. We consider the* coGHIproof$_\ell(S,T)$ *protocol, where $T = \{(x_1,z_1),\dots,(x_t,z_t)\} \subseteq G \times H$.*

1. *The* coGHIproof *protocol is complete.*

2. *Assuming that* Commit *is perfectly hiding, the* coGHIproof *protocol is perfect black-box zero-knowledge against any verifier. Moreover, if* Commit *is a perfectly hiding trapdoor commitment scheme, the* coGHIproof *protocol is perfect black-box straight-line zero-knowledge against any verifier who has the secret key $\mathcal{K}_s^{\mathbf{V}}$ associated to* Commit.

3. *Assuming that* Commit *is a perfectly hiding trapdoor commitment scheme with associated secret key* $\mathcal{K}_s^{\mathbf{V}}$ *of the (honest) verifier, the* coGHIproof *protocol is perfect non-transferable.*

4. *The* coGHIproof *protocol is sound: from any cheating prover* $\mathbf{P}^*$ *who passes the protocol on a set* T *interpolating with* S *in a group homomorphism (i.e.,* f*) with a probability* $\mathsf{Succ}_{\mathbf{P}^*}^{\mathsf{sd}\text{-coGHI}} = \varepsilon$ *and an expert group knowledge of* G*, we can construct an algorithm* $\mathcal{B}$ *which finds a collision on the commitment scheme* Commit *with a probability* $\mathsf{Succ}_{\mathcal{B}}^{\mathsf{com}\text{-}\mathsf{bnd}} \geq \varepsilon(\varepsilon - p^{-\ell})$ *by rewinding* $\mathbf{P}^*$ *once.*

3.1.10 Two-Move variants of Proof Protocols

In [M06] there are also two-move variants presented for proof protocols for GHID and co-GHID. In this work, we will only consider the four-move variants.

3.1.11 Modular Group Generation Decisional Problem

In some settings of the MOVA signature scheme, the prover needs to validate the generated key by performing a proof for the Modular Group Generation Decisional Problem. This can either be done with an interactive protocol, or in a non-interactive way.

Definition 16. (**d-MGGD Problem**, Modular Group Generation Decisional Problem)

Parameters: *An Abelian group* G*, and a positive integer* d*.*

Instance Generation: *A set of values* $S_1 = \{x_1, \ldots, x_s\} \subseteq G$*.*

Problem: *Does* S_1 *modulo* dG *span* G/dG*?*

3.1.12 Proof Protocol for the MGGD Problem

Let G, d be some parameters and S some input of a d-MGGD problem. With the proof protocol for the MGGD problem, a prover can show that a set $S = g_1, ..., g_s$ H-generate G, for any group of order d. This corresponds to show that $\langle S_1 \rangle + dG = G$. The prover needs an expert group knowledge (as defined in Section 3.1.5. Let $\ell \in \mathbb{N}$ be a security parameter. The protocol MGGDproof$_{\text{Ival}}$ is defined as follows:

```
MGGDproof_ℓ(S_1) :
Parameters: $G, d$
Input: $\ell$, $S_1 = \{g_1, \ldots, g_s\} \subseteq G$
1: The prover picks $x_1, \ldots, x_\ell \in_U G$ uniformly at random and computes
   $(\mathsf{com}, \mathsf{dec}) \leftarrow \mathsf{Commit}(x_1, \ldots, x_\ell)$. He sends $\mathsf{com}$ to the verifier.
2: The verifier picks $y_1, \ldots, y_\ell \in_U G$ uniformly at random and sends $y_1, \ldots, y_\ell$
   to the prover.
3: Using expert group knowledge, the prover finds $r_i \in G$, $a_{i,1}, \ldots, a_{i,s} \in \mathbb{Z}_d$
   such that $x_i + y_i = dr_i + \sum_{j=1}^{s} a_{i,j} g_j$ for $i = 1, \ldots, \ell$. He sends the values
   $r_i$'s, $a_{i,j}$'s to the verifier and opens the commitment by sending the $x_i$'s and
   $\mathsf{dec}$.
```

```
4: The verifier checks that $x_i + y_i = dr_i + \sum_{j=1}^{s} a_{i,j} g_j$ holds for $i = 1, \ldots, \ell$
   and that the commitment is correctly opened by checking
   $1 \leftarrow \mathsf{Open}(x_1, \ldots, x_\ell, \mathsf{com}, \mathsf{dec})$. If this is the case, the verifier accepts the
   proof. Otherwise, he rejects it.
```

3.1.13 Properties of Proof Protocol for the MGGD Problem

This proof protocol for the MGGD problem satisfies following security properties (a proof can be found in [M06]):

Theorem 3. *Let G be an Abelian group and d be an integer with smallest prime factor p. We consider the parameters $S_1 = \{g_1, \ldots, g_s\} \subseteq G$ and an integer ℓ. Assuming that the prover has an expert group knowledge of G with S_1, the* MGGDproof$_\ell(S_1)$ *protocol satisfies the following security properties.*

1. *The* MGGDproof *protocol is complete.*

2. *Assuming that* Commit *is a perfectly hiding trapdoor commitment scheme, the* MGGDproof *protocol is perfect black-box straight-line zero-knowledge against any verifier who has the secret key* $\mathcal{K}_\mathrm{s}^{\mathbf{V}}$ *associated to* Commit.

3. *The* MGGDproof *protocol is sound: from any cheating prover* $\mathbf{P}^*$ *who passes the protocol with a set S_1 satisfying $\langle S_1 \rangle + dG \subsetneq G$ with a success probability* $\mathsf{Succ}_{\mathbf{P}^*}^{\mathsf{sd}\text{-}\mathrm{MGGD}} = \varepsilon$, *there exists an algorithm $\mathcal{B}$ which finds a collision on* Commit *with a probability* $\mathsf{Succ}_{\mathcal{B}}^{\mathsf{com}\text{-}\mathsf{bnd}} \geq \varepsilon(\varepsilon - p^{-\ell})$ *by rewinding* $\mathbf{P}^*$ *once.*

3.1.14 Non-Interactive Variant of MGGDproof

There is also a non-interactive variant of the MGGDproof, which has that advantage that the verifier can validate the proof off-line. The proof protocol makes use of the Fiat-Shamir paradigm [FS86].

```
NIMGGDproof_ℓ(S_1)
Parameters: $G, d$
Input: $\ell$, $S_1 = \{g_1, \ldots, g_s\} \subseteq G$
1: The prover picks seedM $\in_U \{0,1\}^{k_m}$ uniformly at random and using the
   pseudorandom generator $\mathsf{GenM}$ produces some challenges $x_1, \ldots, x_\ell$. Then,
   using his expert group knowledge, he finds $r_i \in G$ and $a_{i,1}, \ldots, a_{i,s} \in \mathbb{Z}_d$
   such that $x_i = dr_i + \sum_{j=1}^{s} a_{i,j} g_j$ for $i = 1, \ldots, \ell$. He sends seedM and the
   coefficients $r_i$'s and $a_{i,j}$'s to the verifier.
2: Using $\mathsf{GenM}$, the verifier generates $x_1, \ldots, x_\ell$ from seedM. He checks that
   $x_i = dr_i + \sum_{j=1}^{s} a_{i,j} g_j$ holds for $i = 1, \ldots, \ell$. If this is the case, the verifier
   accepts the proof. Otherwise, he rejects it.
```

3.1.15 Properties of Non-Interactive Variant of MGGDproof

This proof protocol for the MGGD problem satisfies following security properties (a proof can be found in [M06]):

Theorem 4. *Let G be an Abelian group and d be an integer with smallest prime factor p. We consider $S_1 = \{g_1, \ldots, g_s\} \subseteq G$ and an integer ℓ. Assuming that* GenM *is a random oracle and the prover has an expert group knowledge of G with S_1, the* $\mathrm{NIMGGDproof}_\ell(S_1)$ *protocol satisfies the following security properties.*

1. *The* NIMGGDproof *protocol is complete.*
2. *The* NIMGGDproof *protocol is perfect black-box zero-knowledge*[2].
3. *The* NIMGGDproof *protocol is sound: for any set S_1 such that $\langle S_1 \rangle + dG \subsetneq G$, any cheating prover $\mathbf{P}^*$ limited to q_{GenM} queries to* GenM*, has a success probability $\mathsf{Succ}^{\mathsf{sd}\text{-NIMGGD}}_{\mathbf{P}^*} \leq q_{\mathsf{GenM}} \cdot p^{-\ell} + (\#G)^{-\ell}$.*

3.2 MOVA

The MOVA scheme itself is very generic and leaves open many options, such as the setup variant and the chosen homomorphism function. An analysis of all possible variants is beyond the scope of this work. Therefore, we will here only present the generic scheme and discuss some implementation issues in the later sections.

3.2.1 The MOVA Scheme

Domain parameters. We let integers Lkey, Lsig, Icon, Iden be security parameters as well as "group types" for Xgroup and Ygroup. The group types should define what groups and which sizes to use in order to achieve a certain level of security. An optional parameter Ival $\in \mathbb{N}$ is used in Setup Variants 3 and 4 below.

Primitives. We use two deterministic pseudorandom generators GenK and GenS (modelled by some random oracles) which produce elements of Xgroup. We consider a trapdoor commitment scheme Commit[3], for which the associated pair of key $(\mathcal{K}_\mathrm{p}^{\mathbf{V}}, \mathcal{K}_\mathrm{s}^{\mathbf{V}})$ is that of the verifier.

Setup Variant 1. (signer without expert group knowledge)

The signer selects Abelian groups Xgroup and Ygroup of given types together with a secret group homomorphism Hom : Xgroup $\longrightarrow$ Ygroup. He computes the order d of Ygroup. He then picks a random string seedK and computes the Lkey first values $(\mathrm{Xkey}_1, \ldots, \mathrm{Xkey}_{\mathrm{Lkey}})$ from $\mathsf{GenK}(\mathrm{seedK})$ and $\mathrm{Ykey}_j := \mathrm{Hom}(\mathrm{Xkey}_j)$, for $j = 1, \ldots, \mathrm{Lkey}$.

Setup Variant 2. (signer with a Registration Authority and without expert group knowledge)

We use here a Registration Authority (RA) whose role consists of making sure that seedK was randomly selected.

[2]Note that non-interactive zero-knowledge is anyway straight-line.

[3]Note that for the two-move verification protocols we need to specify a trapdoor one-way permutation instead of the trapdoor commitment scheme Commit.

1. The signer selects Abelian groups Xgroup and Ygroup of given type together with a group homomorphism Hom : Xgroup $\longrightarrow$ Ygroup. He computes the order d of Ygroup. She submits his identity Id together with Xgroup, Ygroup and d to RA.
2. RA first checks the identity of the signer and that he did not submit too many registration attempts. He then picks a random string seedK that is sent to the signer together with a signature C of the data
$$(\mathrm{Id}, \mathrm{Xgroup}, \mathrm{Ygroup}, d, \mathrm{seedK}).$$
3. The signer computes the Lkey first values $(\mathrm{Xkey}_1, \ldots, \mathrm{Xkey}_{\mathrm{Lkey}})$ from $\mathsf{GenK}(\mathrm{seedK})$ and $\mathrm{Ykey}_j := \mathrm{Hom}(\mathrm{Xkey}_j)$ for $j = 1, \ldots, \mathrm{Lkey}$.

Setup Variant 3. (signer with an expert group knowledge with validity proof of public key)

In this variant we assume that the signer has an expert group knowledge of Xgroup. It works exactly like in the Setup Variant 1, but the signer can further run a $\mathrm{MGGDproof}_{\mathrm{Ival}}$ in order to validate the public key so that Lkey can be further reduced to the smallest possible one.

Setup Variant 4. (signer with an expert group knowledge with non-interactive validity proof of public key) This variant is the same as Setup Variant 3 except that the signer produces the non-interactive proof $\mathrm{NIMGGDproof}_{\mathrm{Ival}}$.

Public Key. $\mathcal{K}_{\mathrm{p}}^{\mathbf{S}} = (\mathrm{Xgroup}, \mathrm{Ygroup}, d, \mathrm{seedK}, (\mathrm{Ykey}_1, \ldots, \mathrm{Ykey}_{\mathrm{Lkey}}))$ with an optional (Id, C) for the variant 2, an optional Ival for the variants 3,4, and an optional non-interactive proof for the variant 4.

We say that the public key $\mathcal{K}_{\mathrm{p}}^{\mathbf{S}}$ is *valid* if $\{\mathrm{Xkey}_1, \ldots, \mathrm{Xkey}_{\mathrm{Lkey}}\}$ Ygroup-generate Xgroup.

Secret Key. $\mathcal{K}_{\mathrm{s}}^{\mathbf{S}} = \mathrm{Hom}$.

Signature generation. Let m be a message to sign. The signer generates
$$\mathsf{GenS}(m) \rightarrow \mathrm{Xsig}_1, \ldots, \mathrm{Xsig}_{\mathrm{Lsig}}.$$
He then computes $\mathrm{Ysig}_k = \mathrm{Hom}(\mathrm{Xsig}_k)$ for $k = 1, \ldots, \mathrm{Lsig}$. The signature is
$$\sigma = (\mathrm{Ysig}_1, \ldots, \mathrm{Ysig}_{\mathrm{Lsig}}).$$
It is $\mathrm{Lsig} \cdot \log_2 d$ bits long.

Confirmation Protocol. Let (m, σ) be a supposedly valid message-signature pair. Both, the signer and the verifier (signature's recipient) compute the elements $\mathrm{Xkey}_1, \ldots, \mathrm{Xkey}_{\mathrm{Lkey}}$ from the signer's public key. They also generate
$$\mathsf{GenS}(m) \rightarrow \mathrm{Xsig}_1, \ldots, \mathrm{Xsig}_{\mathrm{Lsig}}.$$
The signer playing the role of the prover runs $\mathrm{GHIproof}_{\mathrm{Icon}}$ with the verifier on the set
$$S = \{(\mathrm{Xkey}_j, \mathrm{Ykey}_j) \mid j = 1, \ldots, \mathrm{Lkey}\} \cup \{(\mathrm{Xsig}_k, \mathrm{Ysig}_k) \mid k = 1, \ldots, \mathrm{Lsig}\}.$$

Alternatively, they can run 2-GHIproof$_{\mathrm{Icon}}$ on the same set S.

Denial Protocol. Let (m, σ') be an alleged invalid message-signature pair. We denote

$$\sigma' = (\mathrm{Zsig}_1, \ldots, \mathrm{Zsig}_{\mathrm{Lsig}}).$$

The signer and the verifier compute $\mathrm{Xkey}_1, \ldots, \mathrm{Xkey}_{\mathrm{Lkey}}$ from the public key as well as $\mathsf{GenS}(m) \to \mathrm{Xsig}_1, \ldots, \mathrm{Xsig}_{\mathrm{Lsig}}$. The signer playing the role of the prover run coGHIproof$_{\mathrm{Iden}}$ with the verifier on the sets

$$S = \{(\mathrm{Xkey}_j, \mathrm{Ykey}_j) \mid j = 1, \ldots, \mathrm{Lkey}\},$$

$$T = \{(\mathrm{Xsig}_k, \mathrm{Zsig}_k) \mid k = 1, \ldots, \mathrm{Lsig}\}.$$

Alternatively, they can run 2-coGHIproof$_{\mathrm{Icon}}$ on the same sets S and T.

3.2.2 Security Properties of MOVA

Theorem 5 shows the security properties for the 4-move variant of MOVA and is proved in [M06].

Theorem 5. *Let* $S = \{(\mathrm{Xkey}_1, \mathrm{Ykey}_1), \ldots, (\mathrm{Xkey}_{\mathrm{Lkey}}, \mathrm{Ykey}_{\mathrm{Lkey}})\}$ *and* e *denote the natural logarithm base. Assuming that the signer's public key is valid and* GenS *is a random oracle, the* MOVA *signature scheme with 4-move confirmation and denial protocols satisfies the following security properties.*

1. *The confirmation (resp. denial) protocol is complete.*

2. *Let* p *be the smallest prime factor of* d*. The confirmation (resp. denial) protocol is sound. From any cheating signer* $\mathbf{S}^*$ *who passes the confirmation (resp. denial) protocol on an invalid (resp. valid) signature with a probability* $\mathsf{Succ}^{\mathsf{sd\text{-}con}}_{\mathbf{S}^*} = \varepsilon$ *and an expert group knowledge of* Xgroup*, we can construct an algorithm* $\mathcal{B}$ *which finds a collision on the commitment scheme with a probability*

$$\mathsf{Succ}^{\mathsf{com\text{-}bnd}}_{\mathcal{B}} = \varepsilon(\varepsilon - p^{-\mathrm{Icon}})$$

   (resp. $\varepsilon(\varepsilon - p^{-\mathrm{Iden}})$ *) by rewinding* $\mathbf{S}^*$ *once.*

3. *With a perfectly hiding trapdoor commitment, the confirmation (resp. denial) protocol is perfect black-box straight-line zero-knowledge.*

4. *With a perfectly hiding trapdoor commitment, the confirmation (resp. denial) protocol is perfect non-transferable.*

5. *Consider the* Lsig-S-GHI *problem with the same parameters as for the* MOVA *scheme, i.e.,* $G = \mathrm{Xgroup}$*,* $H = \mathrm{Ygroup}$*. Assume that for any solver* $\mathcal{B}$ *with a given complexity we have*

$$\mathsf{Succ}^{\mathrm{Lsig}\text{-}S\text{-}\mathrm{GHI}}_{\mathcal{B}} \leq \varepsilon.$$

   Then, any forger $\mathcal{F}$ *with similar complexity using* q_S *signing queries and* q_V *queries to the confirmation/denial oracle* Verify *wins the existential forgery game under a chosen-message attack with a probability*

$$\mathsf{Succ}^{\mathsf{ef\text{-}cma}}_{\mathcal{F}} \leq e(1 + q_S)(1 + q_V)\varepsilon.$$

3.2.3 Batch Verification

The MOVA scheme allows the verification of several messages within one protocol execution. This is what we call *batch verification*. The idea is that we put all pairs $(\mathrm{Xsig}_k, \mathrm{Ysig}_k)$ in one single set S. Formally, the protocol is specified as follows:

BatchConf$_\ell$(SIG)
Parameter: $\mathcal{K}_{\mathrm{p}}^{\mathbf{S}}$, ℓ
Input: a set of message-signature pairs $\mathrm{SIG} = \{(m_1, \sigma_1), \ldots, (m_s, \sigma_s)\}$ supposedly valid with respect to the MOVA scheme's public key $\mathcal{K}_{\mathrm{p}}^{\mathbf{S}}$.

1: The prover and the verifier generate $\mathsf{GenS}(m_i) \rightarrow \mathrm{Xsig}_1^i, \ldots, \mathrm{Xsig}_{\mathrm{Lsig}}^i$ and directly retrieve $\mathrm{Ysig}_1^i, \ldots, \mathrm{Ysig}_{\mathrm{Lsig}}^i$ from the signature σ_i for $i = 1, \ldots, s$. They also both retrieve the set
$\mathrm{SK} := \{(\mathrm{Xkey}_1, \mathrm{Ykey}_1), \ldots, (\mathrm{Xkey}_{\mathrm{Lkey}}, \mathrm{Ykey}_{\mathrm{Lkey}})\}$ from $\mathcal{K}_{\mathrm{p}}^{\mathbf{S}}$.
2: For any $i = 1, \ldots, s$, the prover checks whether $\mathrm{Hom}(\mathrm{Xsig}_j^i) = \mathrm{Ysig}_j^i$ for all $j = 1, \ldots, \mathrm{Lsig}$. He puts in memory all indices $(i_1, \ldots, i_k) =: I$ which do not satisfy this property and sends them to the verifier.
3: The prover and the verifier perform $\mathrm{GHIproof}_\ell(\mathrm{SIG}' \cup \mathrm{SK})$, where the set SIG' corresponds to SIG without the message-signature pairs whose index is in I.

If *all* the signatures in SIG are valid, the verifier will accept the proof. But if there is at least one invalid signature, there is no strategy for the prover to make the verifier accept with a probability greater than $p^{-\ell}$, assuming that the commitment scheme's binding property can not be violated.

In the lottery protocol that we propose in Section 4.5 we will make use of the batch verification protocol to efficiently verify *all* the signatures within one protocol execution.

Chapter 4

SMS Lottery

In this section we are analysing different variants of SMS lottery protocols that are based on MOVA.

To simplify this analysis, we have chosen a top-down approach. After defining some terms (Section 4.1) and giving the lottery settings that we consider (Section 4.2), we start with a high-level analysis where we hide some technical details of the MOVA signature scheme (Section 4.3). In that part we assume that we have an ideal implementation of an undeniable signature scheme, where no attacks on the signature scheme itself, such as signature forgery, are possible. In Section 4.4, we focus on the implementation-related topics, such as the required signature size.

4.1 Used Terms

In this subsection we are defining the terms that we are going to use throughout all the work.

The *Player* is the person who wants to play the lotto.

The *Lottery Organization* is the organization that organizes the lottery.

The *Service Provider* is the telephone company that maintains the network for the mobile telephones.

The *Playing Fee* has to be paid by the player when playing. It is either paid directly to the lottery organization or charged by the service provider which transfers the money to the lottery organization.

The *Draw Ticket* is chosen by the player. Normally it contains the chosen numbers and the chosen draw (e.g. the date). The chosen draw can also be contained implicitly (e.g. the next draw).

The *Verification Ticket* is the verification that is sent by the lottery organization to the player (possibly through the intermediate of the service provider).

It normally contains the *Draw Ticket*, the identifier of the player (normally his mobile phone number), a counter that is increased for every ticket and the signature of all this data. The meaning of the verification ticket is to give some sort of proof to the player, that the lottery organization has accepted his ticket. If later the lottery organization claims that the player has not played, he can prove to an authority that the lottery organization has accepted the ticket, and therefore that he is eligible for possible winnings.

The *Verification Interface* is the interface that is provided by the lottery organization to prove the (in)validity of a signature.

Figures 4.1 and 4.2 show the basic playing procedure in the 2-party and 3-party case respectively. In the 2-party case, the *player* chooses the numbers (and some other parameters) and sends them in an SMS to the lottery organization. This information is denoted as *Draw Ticket*. The lottery organization adds some data and signs it. All this data together with the signature are called the *Verification Ticket*. This Verification Ticket is sent back to the player.

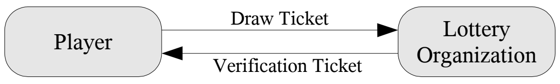


Figure 4.1: Protocol scheme for the 2-party case

In the 3-party case, the *service provider* is interconnected between the player and the lottery organization. Like this, some tasks such as verification of the signatures or administration of the player's accounts can now be assigned to the service provider.

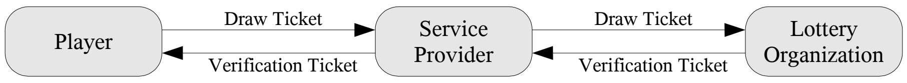


Figure 4.2: Protocol scheme for the 3-party case

4.2 The Lottery Settings

In this work we consider a lottery version that is very similar to the official lottery system in Switzerland[1].

[1]The Swiss Lottery system is described in Appendix A. Our version of the lottery deviates from the Swiss Lottery in the omission of the additional number.

The player has to choose 6 distinct numbers from 1 to 45 per ticket. During the draw, the lottery determines the 6 distinct winning numbers from 1 to 45. For each ticket, the total number of coinciding numbers with the winning numbers determines the winning class:

- 6 correct numbers: Winning class 1
- 5 correct numbers: Winning class 2
- 4 correct numbers: Winning class 3
- 3 correct numbers: Winning class 4

For 2 and less coinciding numbers, the ticket is not eligible for winnings.

4.3 High-Level View of Protocols

4.3.1 Environment

In this high-level analysis of the protocols, we assume that we have a perfect undeniable signature scheme at disposal. *Perfect* means that all the properties stated in Section 2.1.8 are perfectly met, e.g. it is impossible to forge a signature without knowledge of the secret key. Thus we can concentrate on the problems that are independent of the chosen undeniable signature scheme and its parameters.

4.3.2 Main High-Level Options

With a view to propose an *optimal*[2] protocol for SMS lottery that offers cryptographic security, we point out the main options that we have, including the respective advantages and disadvantages.

Implication of the Service Provider

If we imply the service provider in our model we have several advantages: By using the technique of the *batch verification*, the service provider can take an active role by verifying the tickets. Thus, only the players that neither trust the lottery organization, nor the service provider, have to verify their tickets to be sure that the signature is valid. On the other side, the implication of the service provider would require a more complex business model, where the service provider has to be rewarded for its activities. Another disadvantage is, that we have additional security concerns, because the service provider as well could act dishonestly.

Another aspect that we want to discuss here is the question of the payment. Whenever a player plays the lottery, he has to pay the *playing fee*. And whenever he wins in the lottery (even small winnings), this money needs to

[2] *Optimal* means here that we consider the proposed protocol optimal in an industrial context, e.g. the proposed protocol is what we would suggest to a lottery organization that would like to implement SMS lottery with an undeniable signature scheme.

be paid out to him. As the service provider already maintains an account for each player, on which it bills the cost of calls, we could use this account for the lottery. Like this, the player can simply pay the *ticket fee* with the bill from the service provider. Possible winnings could be credited to this account, such that the phone bill would be reduced by the winnings. This method is an additional argument for the implication of the service provider.

Although this method allows an easy way of handling the player's accounts for small winnings, we believe that this is not an adequate procedure for big winnings (e.g. CHF 50.- and more). For these (rather rare) winnings, we suggest that the player transfers the number of his bank account to the lottery organization, whereupon the lottery organization transfers the money on this account.

Verification of the Signatures

To be sure that the lottery organization distributes valid signatures, they should be verified. This verification is normally done by the players. This has the advantage that the players will have the most trust in the system. The disadvantage is, that this needs a sophisticated infrastructure, which should not be too complicated to be used by the players. Even if it is not necessary that *all* the players verify their signature, there should be sufficiently many, such that it is not profitable for the lottery organization to distribute invalid signatures.

If we imply the service provider in the protocol, we can also delegate this task to it. The advantage of this version is, that the verification interface could be implemented in a simpler way, and the players do not need anymore to be annoyed with that task. On the other hand, the players need to trust the service provider. Note that this verification can be done very efficiently for the undeniable signature schemes that offer *batch verification*: An unlimited number of messages can be verified within one single protocol execution of the confirmation protocol.

In a third variant the service provider is in charge of verifying the signatures, but there is still a verification interface for the players. This combines the advantages of the two previous variants: The service provider takes over the charge of verifying the tickets, but if the players have doubts about the trustability of the service provider, they can still verify the tickets themselves. The disadvantage is that the complexity of the verification infrastructure is much higher than in the previous variants.

It has to be specified what happens if there seems to be an invalid signature. Obviously an invalid signature would first of all be a damage to the image of the lottery organization. Additionally, it could even be reasonable that the lottery organization could be accused and an legal inquiry could be activated. If we do not allow validity checking in general, a cheating lottery organization could only be conveyed when a winning ticket is not accepted.

Time Slot for the Verification

The lottery organization can at any time offer an interface that allows the players to verify a signature. The access to the verification interface can possibly help a forger to generate a valid signature. The value of a draw ticket before the draw is limited to the playing fee, because the winning numbers are not yet known. Therefore, a forgery of a draw ticket before the draw seems to be less attractive. In contrast, after the draw, a forger knows which numbers are winning, and therefore a forgery is much more tempting. Thus a limitation of the verification interface to the before-draw period can make sense.

If the service provider is responsible for the verification, we can assume that this is done at once just after the subscription deadline for a particular draw.

Accepted Verification Queries

To make the forgery of a signature more difficult, the lottery organization can limit the allowed verification queries per message. To reduce the number of queries, we propose following measures:

- Only allow the verification of *draw tickets* that have been registered by the lottery organization. If a player wants a message to be verified that has not previously been registered, the lottery organization refuses to perform the verification protocol.
- Only allow the verification of *draw tickets* where the *identifier* corresponds to the player. To realize such an authentication, we can make use of random codes: The lottery organization can, after receiving a request for verifying a signature, send a random code by SMS to the corresponding player. Thereafter the player needs to enter this random code into the verification interface, in order to be able to execute the interactive confirmation / denial protocol.
- Limit the number of verifications to a certain number, e.g. to $1,000$ verifications per player and per draw.

4.3.3 Proposed Protocols

The choices defined above lead to 7 different classes of protocols:

1. Without implication of the service provider, verification only by the players, verification only possible before the draw.
2. Without implication of the service provider, verification only by the players, verification always possible.
3. With implication of the service provider, verification only by the players, verification only possible before the draw.
4. With implication of the service provider, verification only by the players, verification always possible.

5. With implication of the service provider, verification only by the service provider.
6. With implication of the service provider, verification by the service provider *and* the players, verification for the players always possible.
7. With implication of the service provider, verification by the service provider *and* the players, verification for the players only possible before the draw.

The concrete choice of a protocol is always a trade-off between the following properties:

- Difficulty for players to forge a signature.
- Difficulty for the lottery organization or the service provider to act dishonestly (e.g. by distributing invalid signatures without being detected).
- Simplicity for the players.
- Infrastructural cost.

4.3.4 Threat Analysis

Lottery Organization gives an Invalid Signature

The main purpose for signing the *Draw Ticket* by the lottery organization is to give the player some security, that the lottery organization has to accept a winning ticket. Nevertheless, this only works if the lottery organization signs the ticket correctly. Otherwise, the lottery organization could later prove that a ticket is not valid, even if the player has paid for it. In the case where the service provider verifies the validity of the tickets, we can assume that every ticket is verified and the lottery organization will always be detected, if it distributes invalid signatures (as long as there is no conspiracy including the lottery organization *and* the service provider). In the case where the players verify the validity of the signature, they have to have access to the verification interface that is provided by the lottery organization.

The probability that the lottery organization is detected when it distributes invalid signatures is given by the following formula:

$$P_{LOdetected} = 1 - \prod_{i=0}^{c-1} \frac{N-n-i}{N-i}$$

where

- N is the total number of tickets that are played.
- n is the number of invalid signatures that are distributed by the lottery organization.
- c is the number of tickets that are verified by their Players.

For the sake of simplicity, we only consider a single draw and we assume that the tickets that are verified are uniformly distributed among all the tickets.

The following table shows some examples:

N	n	c	$P_{LOdetected}$
10,000	1	100	0.01
10,000	5	100	0.049
10,000	1	500	0.05
10,000	5	500	0.226
10,000	5	1,000	0.41
10,000	10	1,000	0.651
10,000	100	100	0.636
10,000	100	1,000	1

Table 4.1: Detection probability for invalid signatures

It is important to note that at the time when the lottery organization gives an invalid signature, it is not yet known whether the ticket will be a winning ticket or not. Thus, the expected winnings for the lottery organization when giving an invalid signature is at most the price of the lottery ticket. In the same time, this invalid signature will be detected with a probability of c/N. As the expected damage for the image of the lottery organization when being detected is important, the distribution of invalid signatures does not seem to be profitable for the lottery organization.

Service Provider gives an Invalid Signature

In the variants where the service provider is considered as a distinct entity, it is also possible that it does not transmit the draw ticket to the lottery organization. Instead it could directly send an invalid verification ticket and encash the playing fee. The probability that such a fraud is being detected is the same as for the case where the lottery organization gives an invalid signature. Obviously, there has to be a procedure to find out whether the service provider or the lottery organization has cheated.

Detection of Verifiers

In the case where the lottery organization gives an invalid signature, we have assumed that the players that verify the signature are uniformly distributed among all players. This assumption is not always justified, as there might be some players that verify the signatures more often than others. As the lottery organization knows which players are verifying the signature, it could use this knowledge to give invalid signatures to the players that never check their signatures. This can only be avoided if *all* the players verify the signatures from time to time.

Misuse of Knowledge by the Service Provider

The service provider as the intermediary between the players and the lottery organization gains knowledge of the players and respective draw tickets. This knowledge could be misused in several ways:

(a) Privacy of the players Normally the identities of the players that win in the lottery are strictly kept secret, to avoid them from being harassed by invitations to donate or even from criminal threats. As the service provider knows the identities of the players that have won, it could sell this knowledge e.g. to some advertisement companies.

(b) Number analysis The service provider perfectly knows all the numbers that have already been chosen for a particular draw. This knowledge could be used to generate numbers that have not been used yet, to maximize possible winnings. For example, if the main winnings of the lottery are CHF 1,000,000 and the probability to win the main winning is $0.12 \cdot 10^{-6}$ (as in the Swiss Lottery), the expected value of a ticket with a unique number combination is CHF 0.12 (if we only consider the main winnings). If there are already 9 other players that have chosen the same number combination, the expected value is only CHF 0.012.

Note that not only the service provider can misuse the knowledge about the tickets, but any adversary that can listen to the communication (e.g. between the players and the service provider, between the service provider and the lottery organization or between the players and the lottery organization). Therefore it is important that *all* the communication channels are *private*.

4.4 Low-Level View of Protocols

4.4.1 Environment

In the previous part we have assumed that we have a *perfect* implementation of a MOVA scheme. This allowed us to concentrate on issues that are independent of technical details of the implementation. In this part we are diving deeper into details. In order to propose a protocol that can be implemented in the real world, we need to fix some parameters.

4.4.2 Main Low-Level Options

The MOVA scheme itself includes a variety of variants from which every one combines several advantages and disadvantages. When giving a concrete protocol for SMS lottery, we need to determine the optimal variant and the sizes of all the parameters. The possible variants of the MOVA signature include following choices:

Group Homomorphism

For the group homomorphism several propositions have been made, which include characters on $\mathbb{Z}_n^*$, the discrete logarithm in a hidden subgroup of $\mathbb{Z}_n^*$ and RSA. These different variants not only differ in terms of efficiency, but some also have additional properties, such as *Two Levels of Secret*[3]. As the SMS Lottery application does not require any special properties, we will only consider the most efficient[4] variant of these homomorphisms: The character of degree 2 on $\mathbb{Z}_n^*$, which corresponds to the Legendre symbol $\left(\frac{\cdot}{p}\right)$ for a RSA modulus $n = pq$ (p and q are prime).

Size of Xgroup

Our choice of the group homomorphism specifies that $\text{Xgroup} = \mathbb{Z}_n^*$ and $\text{Ygroup} = \{-1, 1\}$. But we still need to specify the order of n in $\mathbb{Z}_n^*$. In the subsequent security analysis we assume that it is computationally infeasible to compute the Legendre symbol $\left(\frac{a}{n}\right)$ on a $a \in \mathbb{Z}_n^*$. Until now it is believed that there is no algorithm that is more efficient to compute the Legendre symbol than by factorizing n. Therefore, we should take an n such that factorizing n is difficult, e.g. a product of two randomly chosen 512-bit prime numbers. As some researchers believe that it might be possible to factorize 1024-bit numbers in the near future, we can also take a 2048-bit value to feel more secure.

Setup Variant

The setup phase aims to provide the lottery organization with a secret key and the players (and possibly the service provider) with the corresponding public key. As already mentioned in Section 3, the public key contains two groups, Xgroup and Ygroup, a seed seedK and a subset $S = (\text{Ykey}_1, ..., \text{Ykey}_{\text{Lkey}})$. The private key consists of a group homomorphism $\text{Hom} : \text{Xgroup} \longrightarrow \text{Ygroup}$. If S does not unambiguously define at most one group homomorphism, the signer could theoretically generate a signature such that it is able to successfully perform both protocols *Confirmation* and *Denial*. To avoid this situation, the setup phase should assure that there exists only one single homomorphism.

Two-Move or Four-Move

The verification protocols (Confirmation and Denial) of the four-move MOVA scheme require four messages to be sent, whereas the two-move MOVA only requires two messages. On the other hand, the four-move version has slightly better security properties, which permits a downsizing of the security parameters Icon and Iden while having the same security level compared to the two-move variant. Additionally, the security of the two-move variant is based on more random oracles, but even the four-move variant can only be proved

[3]The property *Two Levels of Secret*, which is explained in [M06], is not relevant in the context of SMS Lottery.

[4]Here, efficiency is measured in respect of computing time to sign a ticket for a typical computer.

secure in the Random Oracle Model. Therefore, the choice between these two variants is not a crucial one. As we weight the downsizing of the security parameters slightly more important, we will only consider the four-move variant. But also the two-move variant would perfectly match our requirements.

Cryptographic Primitives

The different subprotocols of MOVA make use of several cryptographic primitives, such as *Hash Functions* and *Commitment Schemes*. We have to choose real-world implementation of these idealized primitives.

Encoding of the signature

As the signature will be sent as an SMS and needs possibly to be transcribed manually to another medium, we should choose a signature encoding that is the least error prone as possible. Therefore, the full spectrum of characters can not be exploited, but we will only consider 32 different symbols per character. These symbols are

- All numbers from 1 to 9
- All capital letters: A, B, C, ..., Z but without the letters 'I', 'J', 'O' (as they might be mistaken for other characters)

Like this we have 32 different possibilities per character, which exactly encode 5 bits. We could have a shorter signature if we would consider more characters. But we believe that this inefficiency is justified by the gain of human-readability.

Parameters

The parameters whose size needs to be specified include:

- Lkey: The length of the key
- Lsig: The length of the signature
- Icon: The size of the challenge that needs to be sent in the Confirmation protocol
- Iden: The size of the challenge that needs to be sent in the Denial protocol
- Ival: An additional parameter which is only relevant for the Setup Variants 3 and 4, which set the degree of security for the $\mathrm{MGGDproof}_{\mathrm{Ival}}$ and $\mathrm{NIMGGDproof}_{\mathrm{Ival}}$

4.4.3 Security Analysis

Signature Forgery

In [M06] it is stated that if the success probability of an algorithm of a certain complexity to solve the Lsig-S-GHI problem is limited by ε, then there is no forger $\mathcal{F}$ with similar complexity that can win the existential forgery game under a chosen-message attack with a probability $\mathsf{Succ}_{\mathcal{F}}^{ef-cma} > e(1+q_S)(1+q_V)\varepsilon$, where e is the base of the natural logarithm, q_S the maximum number of queries to the Sign-oracle and q_V the maximum number of queries to the Verify-oracle.

In the *existential forgery* game, the adversary tries to find a pair of an arbitrary message together with a valid signature. An SMS Lottery protocol does not necessarily need to be secure against such an attack, as the probability that a randomly generated message is a valid draw ticket (or even a winning ticket) is very small. We will develop another security model, which allows us to get even shorter signatures.

To model the forgery of a winning ticket, we define the game $\mathsf{Game}^{lot-forg}$:

Definition 17. *$\mathsf{Game}^{lot-forg}$: The game consists of two phases:*

1. *The first phase is called the* Pre-Draw *phase. In this phase the adversary has access to the* GenS*,* Sign *oracles. Also queries of the type (m, sig) to the* Verify *oracle are allowed, as long as she has already asked m to the* Sign *oracle previously. If she wants to go into the second phase, she outputs* okay*.*

2. *The second phase is called the* Post-Draw *phase. The adversary gets a set M_{w} of n different, randomly chosen messages from the message space. She has access to the* GenS *oracle, but not anymore to the* Sign *and* Verify *oracles. In the end, she outputs a pair (m, sig) and she wins the game if $m \in M_{\mathrm{w}}$ and sig is a valid signature for m, and she has not asked the* Sign *query to sign the message m in the* Pre-Draw *phase.*

Note that the game $\mathsf{Game}^{lot-forg}$ is specific to the MOVA signature scheme, as it implies the oracle GenS. It can therefore in general not be applied on other undeniable signature schemes.

The *Pre-Draw* phase corresponds to the time frame before the draw. During this period, an adversary is able to send *draw tickets* to the lottery organization, which will thereafter be signed. Additionally to that, she can access the verification interface to verify whether a given signature is valid or not. Note that she has to pay the *playing fee*, every time she wants a message to be signed. *After* the draw, the adversary learns the numbers that have won. Obviously, her objective is now to find a message that encodes a winning ticket, which she has not asked to the Sign oracle yet. If such a message has already been signed, this does not correspond to a forgery, because she has paid for that ticket and therefore she is entitled to receive the corresponding winnings.

In our game, M_{tot} corresponds to the message space that we consider. That is the set of all the messages that can be queried to the Sign oracle, e.g. all messages that the lottery organization would accept to sign. All the messages that encode *winning* tickets where the identifier is the adversary's, are denoted by M_{w}.

Following theorem is very important for the determination of the signature length:

Theorem 6. *Consider the Lsig-S-GHI problem with the same parameters as for the MOVA scheme, i.e.,* $G = \mathrm{Xgroup}$, $H = \mathrm{Ygroup}$. *Assume that for any solver* $\mathcal{B}$ *with a given complexity, we have*

$$\mathsf{Succ}_{\mathcal{B}}^{Lsig-S-GHI} \leq \varepsilon$$

Then, any forger $\mathcal{F}$ *with similar complexity and using at most* q_s *queries to a signing oracle and at most* q_v *queries to the confirmation/denial oracle, wins the game* $\mathsf{Game}^{lot-forg}$ *with a probability*

$$\mathsf{Succ}_{\mathcal{F}}^{lot-forg} \leq \left(\sum_{i=0}^{q_s} \frac{\binom{N_{\mathrm{w}}}{i}\binom{N_{\mathrm{tot}}-N_{\mathrm{w}}}{q_s-i}}{\binom{N_{\mathrm{tot}}}{q_s}} \cdot (1-\hat{p})^i \right)^{-1} \cdot \frac{\varepsilon}{\hat{p}}.$$

Where N_{tot} *denotes the cardinality of* M_{tot} *and* N_{w} *the cardinality of* M_{w}, *and* $\hat{p}$ *can be any value between* 0 *and* 1.

Proof. Let $\mathcal{F}$ be a forger who succeeds to win $\mathsf{Game}^{lot-forg}$ with a non-negligable probability ε. We will construct an algorithm $\mathcal{B}$ which solves the Lsig-S-GHI problem with

$$S := \{(\mathrm{Xkey}_1, \mathrm{Ykey}_1), ..., (\mathrm{Xkey}_{\mathrm{Lkey}}, \mathrm{Ykey}_{\mathrm{Lkey}})\}$$

using the forger $\mathcal{F}$. At the beginning, $\mathcal{B}$ receives the challenges $x_1, ..., x_{\mathrm{Lsig}} \in \mathrm{Xgroup}$ of the Lsig-S-GHI problem. He picks n messages $m_1, ..., m_n$ uniformly at random from the message space M_{tot}. Then he starts the forger $\mathcal{F}$ and proceeds as follows:

1. During the *Pre-Draw* phase, the oracles are simulated as follows:
   - GenS: If $m \notin M_{\mathrm{w}}$, then he generates an answer of *Type-1*. If $m \in M_{\mathrm{w}}$, he generates an answer of *Type-1* with a probability of $(1-\hat{p})$ and an answer of *Type-2* with a probability of $\hat{p}$. These answers look as follows:
     - ⋆ *Type-1*: $\mathcal{B}$ picks $a_{i,j} \in_U \mathbb{Z}_d$ and $r_i \in_U \mathrm{Xgroup}$ uniformly at random for $i = 1, ..., \mathrm{Lsig}$ and $j = 1, ..., \mathrm{Lkey}$. He answers then
     $$\mathrm{Xsig}_i := dr_i + \sum_{j=1}^{\mathrm{Lkey}} a_{i,j}\mathrm{Xkey}_j \text{ for } i = 1, ..., \mathrm{Lsig}.$$
     - ⋆ *Type-2*: $\mathcal{B}$ picks $a_{i,k} \in_U \mathbb{Z}_d$ and $r_k \in_U \mathrm{Xgroup}$ uniformly at random for $k = 1, ..., \mathrm{Lsig}$. He answers then
     $$\mathrm{Xsig}_i := dr_i + x_i + \sum_{j=1}^{\mathrm{Lkey}} a_{i,j}\mathrm{Xkey}_j \text{ for } i = 1, ..., \mathrm{Lsig}.$$

     For each message, $\mathcal{B}$ keeps the coefficients $a_{i,j}$ and r_i in the memory.
   - Sign: For a message m, if the answer to the GenS query of m was of *type-1*, then it answers with $\mathrm{Ysig}_i := \sum_{j=1}^{\mathrm{Lkey}} a_{i,j}\mathrm{Ykey}_j$ for $i = 1, ..., \mathrm{Lsig}$. Otherwise, it aborts the simulation.

- Verify: $\mathcal{B}$ checks whether m was previously queried to the Sign oracle. If this is the case, $\mathcal{B}$ checks whether the signature is valid and simulates the appropriate protocol. If m has not been queried to the Sign oracle, $\mathcal{B}$ returns null.

If $\mathcal{B}$ receives the output okay, it goes into the second phase.

2. In the *Post-Draw* phase, $\mathcal{B}$ gives M_{w} to $\mathcal{F}$. Then the oracles are simulated as follows:

   - GenS: As in the *Pre-Draw* phase.
   - Sign: $\mathcal{F}$ does not have anymore access to the Sign oracle.
   - Verify: $\mathcal{F}$ does not have anymore access to the Ver oracle.

The success probability for that $\mathcal{B}$ is able to solve the GHID problem is given by

$$\Pr[\mathcal{B} \text{ succeeds}] = \Pr[\mathcal{B} \text{ succeeds}|\mathcal{F} \text{ succeeds}] \cdot \Pr[\mathcal{F} \text{ succeeds}].$$

As we assume that there exists no algorithm $\mathcal{B}$ with a success probability $\mathsf{Succ}_{\mathcal{B}}^{Lsig-S-GHI}$ to solve the GHID problem superior to ε, we have

$$\Pr[\mathcal{B} \text{ succeeds}|\mathcal{F} \text{ succeeds}] \cdot \Pr[\mathcal{F} \text{ succeeds}] \leq \varepsilon.$$

The probability of success for $\mathcal{B}$ to solve the GHID problem given that $\mathcal{F}$ wins the game $\mathsf{Game}^{lot-forg}$ is

$$\Pr[\mathcal{B} \text{ succeeds}|\mathcal{F} \text{ succeeds}] = \left(\sum_{i=0}^{q_s} \frac{\binom{N_{\mathrm{w}}}{i}\binom{N_{\mathrm{tot}}-N_{\mathrm{w}}}{q_s-i}}{\binom{N_{\mathrm{tot}}}{q_s}} \cdot (1-\hat{p})^i\right) \cdot \hat{p}$$

Note that $\hat{p}$ is the probability for that $\mathcal{B}$ succeeds to extract the solution for the GHID problem from the answer of $\mathcal{F}$ and

$$\sum_{i=0}^{q_s} \frac{\binom{N_{\mathrm{w}}}{i}\binom{N_{\mathrm{tot}}-N_{\mathrm{w}}}{q_s-i}}{\binom{N_{\mathrm{tot}}}{q_s}} \cdot (1-\hat{p})^i$$

denotes the probability for that $\mathcal{B}$ does not abort the computation.
Therefore we have:

$$\left(\sum_{i=0}^{q_s} \frac{\binom{N_{\mathrm{w}}}{i}\binom{N_{\mathrm{tot}}-N_{\mathrm{w}}}{q_s-i}}{\binom{N_{\mathrm{tot}}}{q_s}} \cdot (1-\hat{p})^i\right) \cdot \hat{p} \cdot \Pr[\mathcal{F} \text{ succeeds}] \leq \varepsilon$$

$\Pr[\mathcal{F} \text{ succeeds}]$ is bounded by $\mathsf{Succ}_{\mathcal{F}}^{lot-forg}$ and we can derive

$$\mathsf{Succ}_{\mathcal{F}}^{lot-forg} \leq \left(\sum_{i=0}^{q_s} \frac{\binom{N_{\mathrm{w}}}{i}\binom{N_{\mathrm{tot}}-N_{\mathrm{w}}}{q_s-i}}{\binom{N_{\mathrm{tot}}}{q_s}} \cdot (1-\hat{p})^i\right)^{-1} \cdot \frac{\varepsilon}{\hat{p}}$$

□

It is now our objective to determine $\hat{p}$ such that $\mathsf{Succ}_{\mathcal{F}}^{lot-forg}$ is minimized. As it seems to be impossible to find the *optimal* value for $\hat{p}$, we are satisfied with the following approximation

$$\hat{p} = \min\left(\frac{N_{\mathrm{tot}}}{N_{\mathrm{w}}(1+q_s)}, 1\right).$$

.
This choice can be justified by following heuristic argument: For N_{tot} much larger than q_S (which is normally the case), we have the following approximation

$$\left(\sum_{i=0}^{q_s} \frac{\binom{N_{\mathrm{w}}}{i}\binom{N_{\mathrm{tot}}-N_{\mathrm{w}}}{q_s-i}}{\binom{N_{\mathrm{tot}}}{q_s}} \cdot (1-\hat{p})^i\right) \cdot \hat{p} \approx \hat{p}(1-\hat{p}\cdot\frac{N_{\mathrm{w}}}{N_{\mathrm{tot}}})^{q_s}.$$

We can find the value for $\hat{p}$ such that $\alpha(\hat{p}) = \hat{p}(1-\hat{p}\cdot\frac{N_{\mathrm{w}}}{N_{\mathrm{tot}}})^{q_s}$ is maximized by differentiating $\alpha(\hat{p})$ and setting it to zero. Taking into consideration that $\hat{p} \in [0..1]$, we find

$$\hat{p} = \min\left(\frac{N_{\mathrm{tot}}}{N_{\mathrm{w}}(1+q_s)}, 1\right).$$

The same approximation leads us to a very short (and therefore more beautiful) approximation formula for $\mathsf{Succ}_{\mathcal{F}}^{lot-forg}$: If N_{tot} is much larger than q_S, we have the approximation

$$\mathsf{Succ}_{\mathcal{F}}^{lot-forg} \lessapprox (1-\hat{p}\cdot p_m)^{-q_s} \cdot \frac{\varepsilon}{\hat{p}},$$

where $p_m = \frac{N_{\mathrm{w}}}{N_{\mathrm{tot}}}$.

In typical lottery settings, as in the lottery system presented in Section 4.2, experimental results show that we can assume that the optimal value for $\hat{p}$ is $\hat{p} = \frac{1}{p_w + q_S p_w}$ for $q_S > 2^5$. Then we have

$$(1-\hat{p}\cdot p_m)^{-q_s} \cdot \frac{\varepsilon}{\hat{p}} = \left(\frac{q_s}{1+q_s}\right)^{-q_s} \cdot \varepsilon(p_m + q_s p_m) \le e(q_s+1)p_m \cdot \varepsilon,$$

where e is the base of the natural logarithm. This leads us to the approximation

$$\mathsf{Succ}_{\mathcal{F}}^{lot-forg} \lessapprox e(q_s+1)p_m \cdot \varepsilon$$

for q_S reasonably large but enough smaller than N_{tot}.

Signature Length for Security Against Forgery

The analysis of the parameter size strongly relies on the analysis in [M06], where the assumption is made that Xgroup can be adjusted such that $\mathsf{Succ}_{\mathcal{F}}^{\mathrm{Lsig}\text{-}S\text{-}\mathrm{GHI}} \approx d^{-\mathrm{Lsig}}$. This leads to

$$\mathsf{Succ}_{\mathcal{F}}^{lot-forg} \le \left(\sum_{i=0}^{q_s} \frac{\binom{N_{\mathrm{w}}}{i}\binom{N_{\mathrm{tot}}-N_{\mathrm{w}}}{q_s-i}}{\binom{N_{\mathrm{tot}}}{q_s}} \cdot (1-\hat{p})^i\right)^{-1} \cdot \frac{d^{-\mathrm{Lsig}}}{\hat{p}}.$$

We assume that in a context of online verification of a signature a security probability of 2^{-30} is sufficient. This leads us to the formula

$$\mathrm{Lsig} \approx 30 - \log_2\left(\left(\sum_{i=0}^{q_s} \frac{\binom{N_\mathrm{w}}{i}\binom{N_\mathrm{tot}-N_\mathrm{w}}{q_s-i}}{\binom{N_\mathrm{tot}}{q_s}} \cdot (1-\hat{p})^i\right)\cdot\hat{p}\right)$$

In Table 4.2 we give the required MOVA signature size in order to achieve $\mathsf{Succ}_{\mathcal{F}}^{lot-forg} \approx 2^{-30}$ for the lottery settings presented in Section 4.2, where we have $N_\mathrm{tot} = \binom{45}{6} = 8,145,060$ and $N_\mathrm{w} = \sum_{i=3}^{6}\binom{45}{i}\binom{45-6}{6-i} = 194,130$

q_S	Lsig in bits
2^6	33
2^7	34
2^8	35
2^9	36
2^{10}	37
2^{11}	38
2^{12}	39
2^{13}	40
2^{14}	41
2^{15}	42

Table 4.2: Signature size with unforgeability for different values of q_S

As a query to the Sign oracle corresponds to the purchase of a lottery ticket, a high number of queries to the Sign oracle is very costly. Therefore, it is reasonable to assume that no player will buy more than 2^{13} lottery tickets. This allows us to conclude that if we use a fresh key pair for every draw, a signature length of 40 bits is enough to prevent from signature forgery.

Value of Icon and Iden for Soundness

Soundness states that

- For any invalid message-signature pair, the signer can not convince the verifier that it is valid (soundness for confirmation protocol).
- For any valid message-signature pair, the signer can not convince the verifier that it is invalid (soundness for denial protocol).

According to [M06], it is sufficient to set $\mathrm{Icon} = \mathrm{Iden} = 20/\log_2(p)$ to have a soundness probability of 2^{-20} for both, the confirmation and the denial protocol, where p is the smallest prime factor of d (in our case: $p = 2$).

Setup Variants

In the following paragraph, we are analyzing the respective advantages and disadvantages of these 5 setup variants for the context of SMS Lottery.

The Setup Variant 1 has the advantage that the protocol is very simple. On the other hand it requires a relative high security probability of 2^{-80}, which results in a high value for Lkey. In [M06] it is suggested that Lkey should have a size of Lkey $= 81/\log_2(d)$ to have a probability P_{gen} that there exists only a single homomorphism of $P_{gen} \geq 1 - 2^{-80}$.

The Setup Variant 2 reduces the value of Lkey drastically as there is no offline exhaustive search possible anymore. In [M06] the size of Lkey is suggested to be $21/\log_2(d)$ to have an *online* security probability of $P_{gen} \geq 1 - 2^{-20}$. The obvious disadvantage of that variant is the need of a trusted *registration authority*. It seems to be very difficult to find such an authority that is in the same time trusted by the players, and also willing to take this task.

The Setup Variant 3 again reduces the value of Lkey to the smallest possible value by the application of the interactive protocol MGGDproof$_{\mathrm{Ival}}$. The players can be convinced that there is only one single homomorphism, independently of the value of Lkey. The parameter Ival becomes the only crucial value to determine the validity of the public key. To have a security probability of 2^{-20}, [M06] suggest that we should have Ival $= 20/\log_2(p)$, assuming that no efficient attacker breaks the computationally binding property of Commit. Nevertheless, Setup Variant 3 requires that the lottery organization provides a corresponding interface, and it does not seem to be very practicable that the players execute the protocol. In addition, the provided interface could be used for forgery attacks.

The advantages of the Setup Variant 4 are that it does not imply any minimum value for Lkey and there is no need to execute an interactive protocol. On the other hand, we need a higher value for Ival. [M06] suggests that Ival $= 80/\log_2(p)$.

Note that for the Setup Variants 3 and 4, we impose that the signer has *Expert Group Knowledge*, which is a restriction on the *group homomorphism*. The group homomorphism that we consider for SMS lottery is the character of order 2, which offers Expert Group Knowledge. Therefore, this is no problem.

4.5 Proposed Lottery Protocol

In this section we propose a protocol for secure SMS lottery that we believe is *optimal* for a commercial success. We are taking into consideration the general protocols of Section 4.3.3, the analysis of the choice of the MOVA variant of Section 4.4.2 and the security analysis of Section 4.4.3. But also business aspects need to be considered, such as the expected acceptance by the players and the cost for the infrastructure. Obviously, there is no solution that is optimal for *all* the aspects. But we believe that the following protocol is a reasonable compromise which offers

- High security
- High acceptance from the players
- Maintainable infrastructural cost

The choices we make are as follows:

- We take the general protocol 7 from Section 4.3.3, i.e. the protocol with the implication of the service provider, which is in charge of the verification of the verification ticket, and where the players have additionally the possibility to verify their ticket, but only *before* the draw.
- As group homomorphism we (as already stated in Section 4.4.2) take the characters of degree 2 on $\mathbb{Z}_n^*$, for n the product of two large prime numbers p and q. This corresponds to the Legendre symbol with respect to one of the prime numbers.
- As Setup Variant we take number 4, where the signer produces the non-interactive proof $\text{NIMGGDproof}_{\text{Ival}}$ in order to validate the public key.
- As values for Ival we take 80.
- As already mentioned in Section 4.4.2 we take the four-move variant.
- As a character encoding for the signature we take the proposition made in Section 4.4.2.
- As Xgroup we take $\mathbb{Z}_n^*$ for an n as a product of two randomly chosen 1024-bit prime numbers.
- As a value for Lsig (the length of the signature) we take 40.
- As values for Icon and Iden we take 20.

More formally, our protocol consists of four subprotocols:

1. **Setup**: The public key and secret key are generated.
2. **Playing**: The player sends the chosen numbers to the service provider, which transfers it to the lottery organization. The lottery organization signs it and sends it back.
3. **ServiceProviderVerify**: The service provider verifies all the signatures once per draw.
4. **PlayerVerify**: The player verifies the signature whenever he wants, but before the draw.

4.5.1 The Protocol Setup

In the initialization phase, the lottery organization generates the keys and the validity proof as follows:

1. Generate two *secure* random prime numbers p and q with a length of 1024 bits. Therefore we have:
   - $n = p \cdot q$
   - $\text{Xgroup} = \mathbb{Z}_n^*$

- Ygroup $= \{-1, 1\}$
- $d = 2$
- Hom $=$ Xkey $\rightarrow$ Ykey, Hom$(a) := \left(\frac{a}{p}\right)$, where $\left(\frac{a}{p}\right)$ denotes the Legendre symbol

2. Choose a random seed *seedk* and take the first 2 outputs from the pseudo-random generator $\mathsf{GenK}(\text{seedK})$ to compute $\{\text{Xkey}_1, \text{Xkey}_2\}$. Repeat this step until the received $\{\text{Xkey}_1, \text{Xkey}_2\}$ defines exactly *one* group homomorphism.[5]

3. Apply the homomorphism on $\{\text{Xkey}_1, \text{Xkey}_2\}$ to compute $\{\text{Ykey}_1, \text{Ykey}_2\}$.

4. Publish the public key $\mathcal{K}_\text{p} = (\text{Xgroup}, \text{Ygroup}, d, \text{seedK}, (\text{Ykey}_1, \text{Ykey}_2))$.

5. Publish a non-interactive proof $\text{NIMGGDproof}_{\text{Ival}}$ for the validity of the public key with the parameter Ival $= 80$.

4.5.2 The Protocol Playing

Figure 4.3 shows the playing procedure. The steps are as follows:

1. The player chooses the numbers for the lotto and writes them together with the number of the draw on an SMS, as for example:

   ```
   0162:01 09 12 15 27 42
   ```

   where `0162` is the number of the draw. He sends this SMS to a dedicated number.

2. The service provider receives the SMS and checks if it contains the right format. If not, it sends an error message per SMS and the protocol is aborted. Otherwise it appends the identifier of the player and generates the message for the lottery organization, which could look as follows (in an XML way):

   ```
   <id>+41763882500</id>
   <drawN>0162</drawN>
   <num>01 09 12 15 27 42</num>
   ```

   This data is sent to the lottery organization, for instance through Internet.

3. The lottery organization has to store this information in the database, together with a counter n which is incremented every time a player plays for the same draw. It then generates a signature for all this information by using the function **genSign** which is described below. The size of the signature is 40 bits, which is equivalent to 8 characters by encoding in the way we described in Section 4.4.2. Then it sends all this data to the service provider:

[5] The conditions for $\{\text{Xkey}_1, \text{Xkey}_2\}$ to define exactly *one* homomorphism require non-trivial mathematics and are stated in [M06].

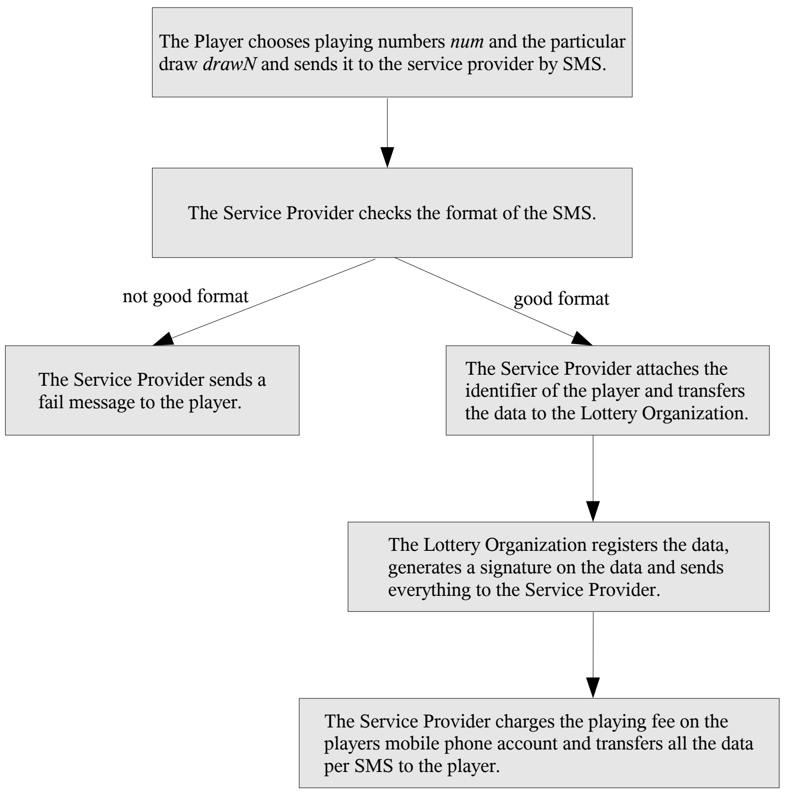

The Player chooses playing numbers *num* and the particular draw *drawN* and sends it to the service provider by SMS.
The Service Provider checks the format of the SMS.
not good format
good format
The Service Provider sends a fail message to the player.
The Service Provider attaches the identifier of the player and transfers the data to the Lottery Organization.
The Lottery Organization registers the data, generates a signature on the data and sends everything to the Service Provider.
The Service Provider charges the playing fee on the players mobile phone account and transfers all the data per SMS to the player.

Figure 4.3: Protocol Playing

```
<id>+41763882500</id>
<drawN>0162</drawN>
<num>01 09 12 15 27 42</num>
<n>1</n>
<sig>E14YGW1R</sig>
```

4. The service provider stores this information and charges the mobile phone account of the corresponding player with the playing fee and transmits all the data as SMS to the player:

```
This is the receipt for player +41763882500,
draw 162(1) Your numbers are 01 09 12 15 27 42,
verification receipt:  E14YGW1R
```

The function genSign

The function **genSign** takes as input an arbitrary message and outputs the according signature. To do so, it first applies a hash function on the message to generate a seed. This seed is used in the pseudorandom generator GenS to generate Lsig elements of Xgroup (in our case, Lsig = 40) $\{\mathrm{Xsig}_1, ..., \mathrm{Xsig}_{\mathrm{Lsig}}\}$. The signature consists of $\{\mathrm{Lsig}_1, ..., \mathrm{Ysig}_{\mathrm{Lsig}}\}$, such that $\mathrm{Ysig}_i = \mathrm{Hom}(\mathrm{Xsig}_i)$ for all $i = 1, ..., \mathrm{Lsig}$.

4.5.3 The Protocol ServiceProviderVerify

Just after the deadline for a particular draw, the service provider checks all the signatures with the interactive *Batch Verification* protocol, which is described in Section 3.2.3.

4.5.4 The Protocol PlayerVerify

Before the draw of the winning numbers, the lottery organization maintains a verification interface, where skeptical players have the possibility to verify their signature. As there is a pretty large amount of data that needs to be exchanged, an SMS is not the appropriate medium to do so. Therefore, we suggest that there is an open-source tool that can be downloaded and that executes the interactive verification protocol directly with the lottery organization through Internet. The player only needs to enter following information:

- the identifier
- the draw number
- the counter
- the chosen playing numbers
- the signature

The lottery organization thereafter sends a random code (of for example 10 characters) per SMS to the player with the corresponding identity. After entering this random code, the program executes the *confirmation* or the *denial* protocol (dependent on whether the lottery organization claims that the ticket is valid or not), and finally outputs one of six possible values:

1. The signature is valid.
2. The signature is invalid.
3. The lottery organization claims that the signature is valid, but the proof is incorrect.
4. The lottery organization claims that the signature is not valid, but the proof is incorrect.
5. The corresponding ticket is not registered.
6. The protocol could not be executed due to network problems (or others).

4.5.5 Pay off Winnings

If a player wins, he is entitled to obtain the winnings. Depending on the size of this amount, we propose two ways to pay out the money:

1. If the amount is smaller than a certain threshold (let us say 50.- CHF), the money is directly paid out by the service provider onto the players mobile phone account.
2. If the amount is bigger than the threshold, the money needs to be requested at the lottery organization, which thereafter transfers the money onto the player's bank account.

4.6 Alternatives to Undeniable Signature Schemes

4.6.1 Conventional Digital Signature Schemes

As an alternative to protocols using undeniable signature schemes we propose a protocol that is using conventional digital signature schemes. Assume a secure digital signature scheme (such as RSA) consisting of following algorithms:

- a probabilistic key generation algorithm $(\mathcal{K}_\text{s}, \mathcal{K}_\text{p}) \leftarrow \mathsf{KGen}(1^k)$ for a security parameter k
- a signature algorithm $sig \leftarrow \mathsf{Sign}(m, \mathcal{K}_\text{s})$
- a verification algorithm $v \leftarrow \mathsf{Verify}(m, sig, \mathcal{K}_\text{p})$ for $v \in \{0, 1\}$.

Initialization

The lottery organization determines $\mathcal{K}_\text{s}$ and $\mathcal{K}_\text{p}$ with $\mathsf{KGen}(1^k)$. The public key is published.

Playing

The playing protocol consists of following steps:

1. The player sends the SMS containing the draw number and the chosen playing numbers to the dedicated number.
2. The service provider appends the identifier and transfers all the data to the lottery organization.
3. The lottery organization appends the counter n (as for the protocol with undeniable signatures) and signs all this data with $Sign(data, \mathcal{K}_\mathrm{s})$. It appends sig to the message and returns it to the service provider.
4. The service provider charges the player's account and transfers the message to the player.

Verification

Every player is now able to verify whether the signature is valid or not with the aid of the protocol Verify and the public key $\mathcal{K}_\mathrm{p}$. To do so, he has to execute a dedicated application, e.g. on a PC.

Advantages and Disadvantages compared to the Protocol with Undeniable Signatures

The key advantage for the version with conventional signatures is, that there is no need for interactive proofs. On the other hand, the signature length has to be much higher. According to our knowledge, the signature scheme with the shortest signature that can be regarded as secure, still requires a signature length of 160 bits ([BLS01]). If we choose a user-friendly version where we limit the number of bits we encode to 5 per character, the signature has a total length of 32 characters. A according verification SMS could look as follows:

```
This is the receipt for player +41763882500,
draw 162(1) Your numbers are 01 09 12 15 27 42,
verification receipt:  XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
```

Where the `X` represents the signature.

Alternatively the signature could be sent per email. But this requires a subscription process, where the players have to communicate their email address, unless this information is stored anyway at the service provider.

4.6.2 Message Authentication Codes

The application of Message Authentication Codes (MACs) is inappropriate, as the verification process requires the knowledge of the secret key. If the secret key is disclosed, everybody can forge signatures, and if it is not disclosed, the verification of the signature by the players is not possible.

Chapter 5

SMS Lottery Simulator

The *SMS Lottery Simulator* is a Java application that aims to prove the concept of an SMS lottery using the MOVA signature scheme to sign the tickets. It implements all the important parts of the proposed protocol of Section 4.5, such as batch verification and key verification. The parameters, such as the key length, are identical to the derived parameters of the security analysis. Even with a test computer, a mobile personal computer with typical settings, all the computations are performed within a reasonable amount of time. Thus, the implementation clearly shows, that a realization of SMS lottery with the MOVA scheme is technically fully feasible.

The SMS Lottery Simulator implements the protocol that we proposed in Section 4.5 for the lottery system presented in Section 4.2. Recall that our lottery system consists of choosing 6 numbers in the range of 1 to 45. During the draw, the lottery organization also determines a set of 6 numbers. The tickets, on which at least 3 numbers coincide with the numbers of the lottery, entitle the owner of the ticket to encash the winnings. The more numbers coincide, the higher are the winnings.

The SMS Lottery Simulator consists of four independent parts:

- The *Key Generator* generates a public key and a secret key, and writes them into two separate files. Additionally, it generates a validity proof, that can be verified by the recipients of the public key.
- The *Player* implements all the possibilities that a player of an SMS lottery has: He can play the lottery by sending an SMS and he can verify signatures through the verification interface. For the verification of the signature, he needs to have access to the public key of the lottery organization.
- The *Service Provider* is in the middle between the players and the lottery organization. All the communications go through it. When a player sends a ticket to play the lottery, the service provider stores the verification ticket from the lottery in its database. By clicking on *Batch Verification* it performs the according protocol to verify, whether all the signatures from the lottery organization are valid. The service provider is also in charge of the administration of the accounts of the players. Thus, it

subtract the *ticket fee* from the player's account whenever he plays, and it adds the winning money if a player has won. To enable the batch verification, the service provider needs to have access to the public key.

- The *Lottery Organization* checks incoming lottery tickets, and signs them if they match the correct format (e.g. the draw number is valid, the playing numbers are in the right range, etc.). It also provides two verification interfaces, one for the players and one for the service provider (for the batch verification). As it only accepts to execute the confirmation (or the denial protocol) if the according ticket has already been signed, it also needs to keep track of all the tickets that have been signed. To test the correctness of the protocols, there is also the possibility to enforce the lottery organization to act dishonestly: In this case, instead of generating valid signatures, it signs the tickets with a random value. The lottery organization needs access to the secret key.

Note that when we write *Player*, *Service Provider* and *Lottery Organization* in capital letters, we refer to the corresponding computer programs. On the other hand we refer to the entities of our real-world system by writing *player*, *service provider* and *lottery organization* in small letters.

The *Player*, *Service Provider* and the *Lottery Organization* programs can be deployed on independent computers. They only need a network connection allowing socket communication between each other, to execute the different protocols.

Figure 5.1 shows what the simulator looks like when all three parts are running on the same computer.

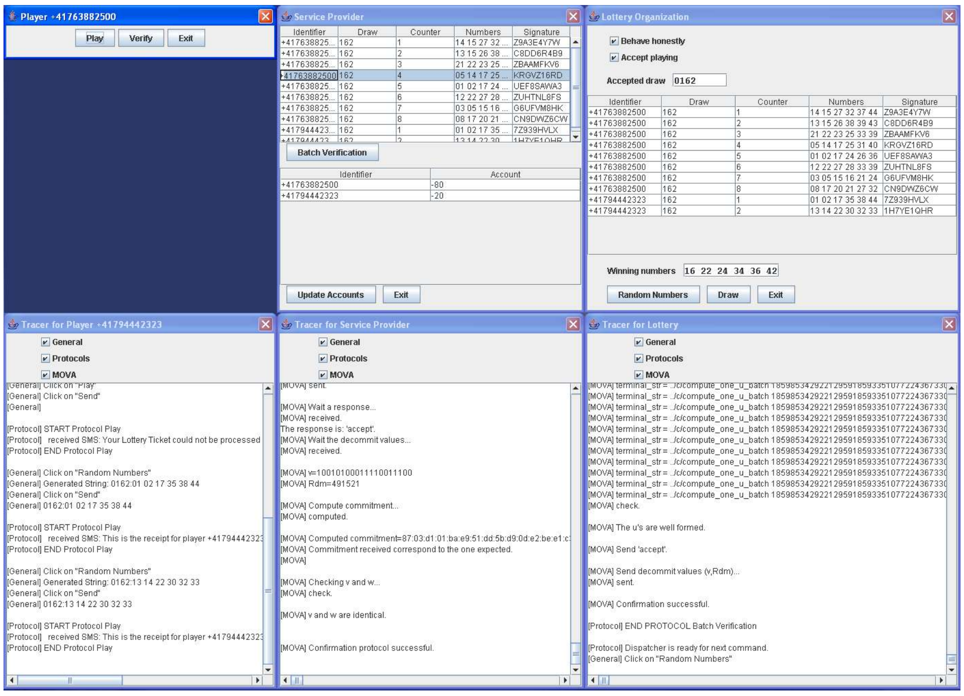
In Section 5.1 we explain the installation and use of the different components. A detailed documentation of the Java classes can be found in Section 5.2 and the C programs are treated in Section 5.3. The protocols that are used for data exchange between the independent programs are discussed in 5.4. In Section 5.5 finally we have a closer look at some parts of the program, such as how the confirmation protocol is implemented.

5.1 Documentation

5.1.1 Requirements and Installation

The SMS Lottery Simulator requires

- A PC with a Linux operating system working in the graphic mode
- The Java Runtime Environment
- A web browser to view the *Javadoc* documentation of the java classes
- A network connection allowing socket communication between the computers, if the *Player*, the *Service Provider* and the *Lottery Organization* are not deployed on the same computer



Identifier	Draw	Counter	Numbers	Signature
+417638825...	162	1	14 15 27 32 ...	Z9A3E4Y7W
+417638825...	162	2	13 15 26 38 ...	C8DD6R4B9
+417638825...	162	3	21 22 23 25 ...	ZBAAMFKV6
+41763882500	162	4	05 14 17 25 ...	KRGVZ16RD
+417638825...	162	5	01 02 17 24 ...	UEF8SAWA3
+417638825...	162	6	12 22 27 28 ...	ZUHTNL8FS
+417638825...	162	7	03 05 15 16 ...	G6UFVM8HK
+417638825...	162	8	08 17 20 21 ...	CN9DWZ6CW
+417944423...	162	1	01 02 17 35 ...	7Z939HVLX
+417944423...	162	2	13 14 22 30	1H7YE1QHR

Identifier	Account
+41763882500	-80
+41794442323	-20

[✓] Behave honestly
[✓] Accept playing
Accepted draw 0162

Identifier	Draw	Counter	Numbers	Signature
+41763882500	162	1	14 15 27 32 37 44	Z9A3E4Y7W
+41763882500	162	2	13 15 26 38 39 43	C8DD6R4B9
+41763882500	162	3	21 22 23 25 33 39	ZBAAMFKV6
+41763882500	162	4	05 14 17 25 31 40	KRGVZ16RD
+41763882500	162	5	01 02 17 24 26 36	UEF8SAWA3
+41763882500	162	6	12 22 27 28 33 39	ZUHTNL8FS
+41763882500	162	7	03 05 15 16 21 24	G6UFVM8HK
+41763882500	162	8	08 17 20 21 27 32	CN9DWZ6CW
+41794442323	162	1	01 02 17 35 38 44	7Z939HVLX
+41794442323	162	2	13 14 22 30 32 33	1H7YE1QHR

Winning numbers 16 22 24 34 36 42

[✓] General
[✓] Protocols
[✓] MOVA

```
[General] Click on "Play"
[General] Click on "Send"
[General]

[Protocol] START Protocol Play
[Protocol]   received SMS: Your Lottery Ticket could not be processed
[Protocol] END Protocol Play

[General] Click on "Random Numbers"
[General] Generated String: 0162:01 02 17 35 38 44
[General] Click on "Send"
[General] 0162:01 02 17 35 38 44

[Protocol] START Protocol Play
[Protocol]   received SMS: This is the receipt for player +41794442323
[Protocol] END Protocol Play

[General] Click on "Random Numbers"
[General] Generated String: 0162:13 14 22 30 32 33
[General] Click on "Send"
[General] 0162:13 14 22 30 32 33

[Protocol] START Protocol Play
[Protocol]   received SMS: This is the receipt for player +41794442323
[Protocol] END Protocol Play
```

[✓] General
[✓] Protocols
[✓] MOVA

```
[MOVA] sent.

[MOVA] Wait a response...
[MOVA] received.
The response is: 'accept'.
[MOVA] Wait the decommit values...
[MOVA] received.

[MOVA] v=10010100011110011100
[MOVA] Rdm=491521

[MOVA] Compute commitment...
[MOVA] computed.

[MOVA] Computed commitment=87:03:d1:01:ba:e9:51:dd:5b:d9:0d:e2:be:e1:c
[MOVA] Commitment received correspond to the one expected.
[MOVA]

[MOVA] Checking v and w...
[MOVA] check.

[MOVA] v and w are identical.


[MOVA] Confirmation protocol successful.
```

[✓] General
[✓] Protocols
[✓] MOVA

```
[MOVA] terminal_str = ../c/compute_one_u_batch 1859853429221295918593351077224367330
[MOVA] terminal_str = ../c/compute_one_u_batch 1859853429221295918593351077224367330
[MOVA] terminal_str = ../c/compute_one_u_batch 1859853429221295918593351077224367330
[MOVA] terminal_str = ../c/compute_one_u_batch 1859853429221295918593351077224367330
[MOVA] terminal_str = ../c/compute_one_u_batch 1859853429221295918593351077224367330
[MOVA] terminal_str = ../c/compute_one_u_batch 1859853429221295918593351077224367330
[MOVA] terminal_str = ../c/compute_one_u_batch 1859853429221295918593351077224367330
[MOVA] terminal_str = ../c/compute_one_u_batch 1859853429221295918593351077224367330
[MOVA] terminal_str = ../c/compute_one_u_batch 1859853429221295918593351077224367330
[MOVA] terminal_str = ../c/compute_one_u_batch 1859853429221295918593351077224367330
[MOVA] terminal_str = ../c/compute_one_u_batch 1859853429221295918593351077224367330
[MOVA] check.

[MOVA] The u's are well formed.

[MOVA] Send 'accept'.

[MOVA] Send decommit values (v,Rdm)...
[MOVA] sent.

[MOVA] Confirmation successful.

[Protocol] END PROTOCOL Batch Verification

[Protocol] Dispatcher is ready for next command.
[General] Click on "Random Numbers"
```

Figure 5.1: Screen of the SMS Lottery Simulator

If you do not want to make use of the Key Generation program, you do not need to install the program on your hard disk, you can simply start the Player, the Service Provider and the Lottery Organization from the CD-ROM. To make use of the Key Generation program, you can copy all the directory (including the subdirectories) onto your hard disk. Then simply change your directory into `SMSLotteryAppl/java` and run the programs as stated in the next section.

If the auxiliary programs in the directory `SMSLotteryAppl/c` do not execute correctly, you can also recompile them to make them compliant with your Linux system. To do so, simply remove all the executable files and run `make` (this requires Gnu Make).

The Java files can be recompiled by executing the program `compile` in the directory `SMSLotteryAppl/java`.

5.1.2 Starting

All the programs can be started from the directory `SMSLotteryAppl/java`. For a quick starting on *one* computer, you can simply start the following scripts:

- `runLotteryOrganization`
- `runServiceProvider`
- `runPlayer`

They access the already provided key files `sms.pmova` and `sms.smova`. To generate new keys `sms.pmova` and `sms.smova`, you can execute `generate_keys`.

The following subsections explain how to exploit the full range of possibilities by using the command-line parameters.

Note, that you always have to start the programs in the following order: 1. Lottery Organization, 2. Service Provider and 3. Player. This is due to the fact that when the Service Provider is startet, it first tries to contact the Lottery Organization. If this is not possible, it crashes. The same applies for the Player with respect to the Service Provider.

Key Generator

The Key Generator can be started with following command:

```
java -cp bin ch/epfl/lasec/mova/KeyGeneration [parameters]
```

The parameters include:

- -pk ⟨*file*⟩: The filename to which the public key is written. The default value is *default.pmova*.
- -sk ⟨*file*⟩: The filename to which the secret key is written. The default value is *default.smova*.
- -v: Generates a validity proof for the key and writes it into the file containing the public key.
- -help: Outputs a help message.

Note that only for keys that were generated with the parameter `-v`, the validity can be verified by the Player, Service Provider and Lottery Organization. E.g. with the command

```
java -cp bin ch/epfl/lasec/mova/KeyGeneration -pk sms.pmova
   -sk sms.smova -v
```

the files `sms.pmova` and `sms.smova` are created. `sms.pmova` contains the public key and a validity proof for the public key. `sms.smova` contains the secret key.

Player

The Player can be started with following command:

```
java -cp bin ch/epfl/lasec/sms_lottery/Player [parameters]
```

The parameters include:

- -id ⟨*identifier*⟩ : The identifier of the player, should be a phone number. The default value is *+41763882500*.

- -p ⟨*port*⟩: The port of where the Player is listening. Choose a free port and do not forget to turn off the firewall on that port. The default value is *8081*.
- -sp ⟨*host*⟩ ⟨*port*⟩: The host and port where the Service Provider is listening on. The default values are *localhost* for *host* and *8082* for *port*.
- -lo ⟨*host*⟩ ⟨*port*⟩: The host and port where the Lottery Organization is listening on. The default values are *localhost* for *host* and *8083* for *port*.
- -pk ⟨*file*⟩: The filename from which the public key is loaded. The default value is *default.pmova*.
- -help: Outputs a help message.

E.g. with the command

```
java -cp bin ch/epfl/lasec/sms_lottery/Player -id +41774881234
   -p 8088 -sp lasecpc15 8033 -pk example.pmova
```

a new Player instance is started for a player with the identity *+41774881234*, which listens on the port *8088*. It assumes the Service Provider to be running on the host *lasecpc15* and listening on the port *8033*. As the *host* and *port* of the Lottery Organization are not specified, it takes the default values (*localhost* for the host and *8083* for the port). The public key is loaded from *example.pmova*.

Service Provider

The Service Provider can be started with following command:

```
java -cp bin ch/epfl/lasec/sms_lottery/ServiceProvider [parameters]
```

The parameters include:

- -p ⟨*port*⟩: The port where the Service Provider is listening. Choose a free port and do not forget to turn off the firewall on that port. The default value is *8082*.
- -lo ⟨*host*⟩ ⟨*port*⟩: The host and port where the Lottery Organization is listening on. The default values are *localhost* for *host* and *8083* for *port*.
- -pk ⟨*file*⟩: The filename from which the public key is loaded. The default value is *default.pmova*.
- -help: Outputs a help message.

E.g. with the command

```
java -cp bin ch/epfl/lasec/sms_lottery/ServiceProvider -p 8100
   -lo lasecpc18 8135
```

the Service Provider is started and listens on the port *8100*. It assumes the Lottery Organization to be running on the host *lasecpc18* and listening on the port *8135*. The public key is loaded from *default.pmova*.

Lottery Organization

The Lottery Organization can be started with following command:

```
java -cp bin ch/epfl/lasec/sms_lottery/LotteryOrganization [params]
```

The parameters include:

- -p ⟨*port*⟩: The port where the Lottery Organization is listening. Choose a free port and do not forget to turn off the firewall on that port. The default value is *8083*.
- -pk ⟨*file*⟩: The filename from which the public key is loaded. The default value is *default.pmova*.
- -sk ⟨*file*⟩: The filename from which the secret key is loaded. The default value is *default.smova*.
- -help: Outputs a help message.

E.g. with the command

```
java -cp bin ch/epfl/lasec/sms_lottery/LotteryOrganization
   -pk test.pmova -sk test.smova
```

the Lottery Organization is started and listens on the port *8083* (per default). The public key is loaded from *test.pmova* and the secret key is loaded from *test.smova*.

5.1.3 Using the Player

When starting the Player, two windows appear:

- The main windows (see Figure 5.2)
- The tracer window (see Figure 5.3)

Additionally, if the command-line option `-v` is set, a warning window will appear if the verification of the public key has failed.

By clicking on the button *Play* in the *main window*, the user can start the *playing interface*. The button *Verify* starts the *verification interface*. By clicking on *Exit*, the Player quits.

The *Tracer Window* is showing in real time, what the program is doing. All the output messages belong to one out of three categories:

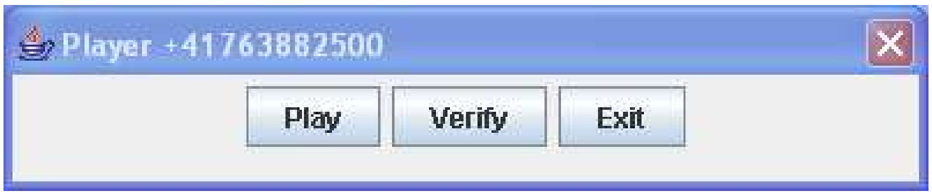


Figure 5.2: Main window of the Player

- General: All the messages that are neither concerning the protocols nor the MOVA signature scheme
- Protocol: All the messages that are concerning the protocols (see Section 5.4 for more information about the protocols)
- MOVA: All the messages that are concerning the different algorithms of the MOVA signature scheme

By switching the check-boxes *General*, *Protocols* and *MOVA*, the output of the respective categories can be enabled / disabled.

Playing Interface

The advantage of an SMS Lottery is that the player can play from everywhere he likes. The only tool he needs is a mobile phone. To make the program as realistic as possible, the *Playing Interface* simulates a mobile phone, from which the user can send SMSs to the Lottery Organization and receive SMSs.

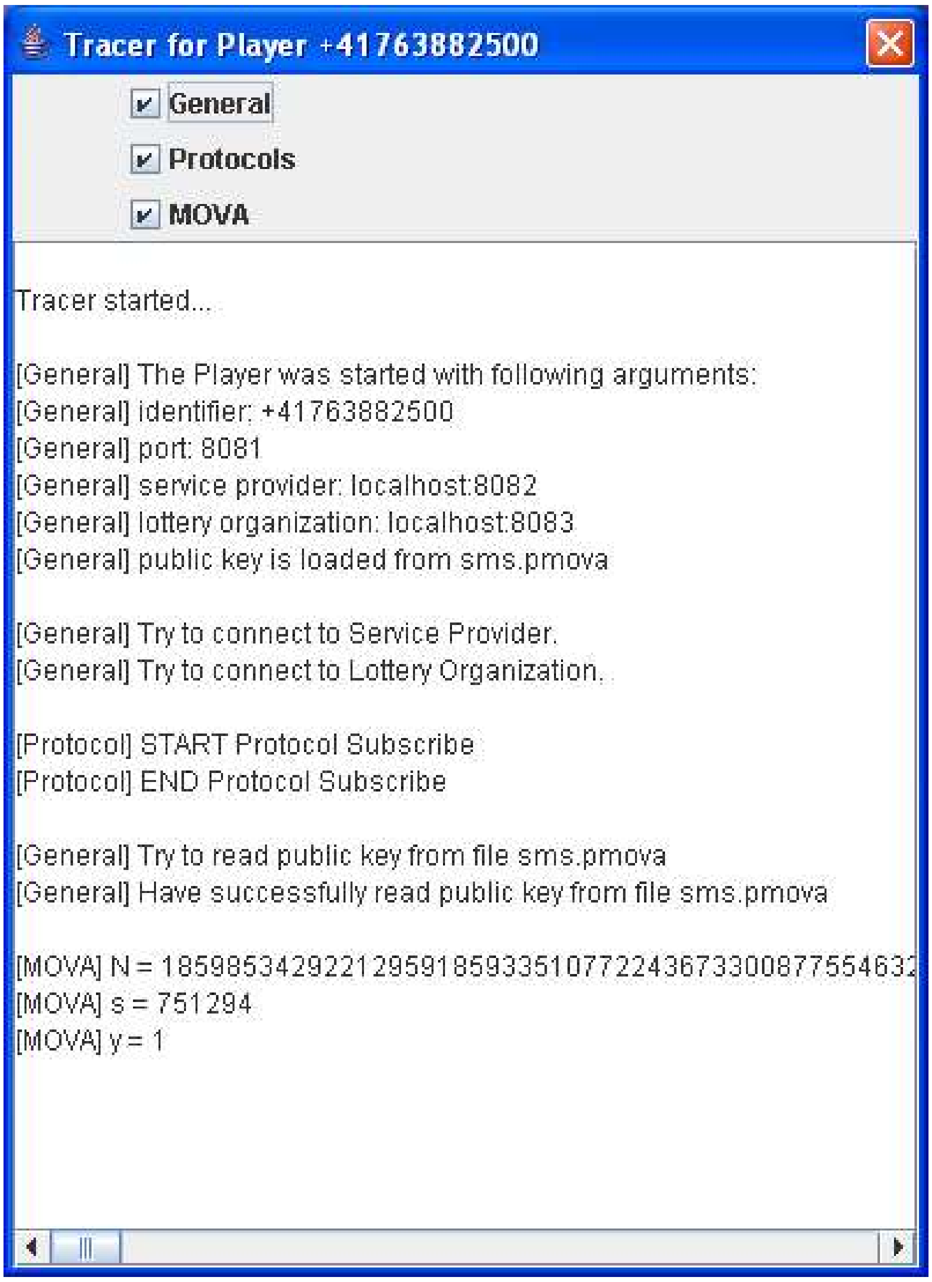
Figure 5.4 shows the playing interface (A) when newly created, (B) after having entered the *draw number* and the *playing numbers* and (C) after having received the confirmation from the Lottery Organization.

To play the lottery, the user has to enter a text in the form

```
DDDD:NN NN NN NN NN NN
```

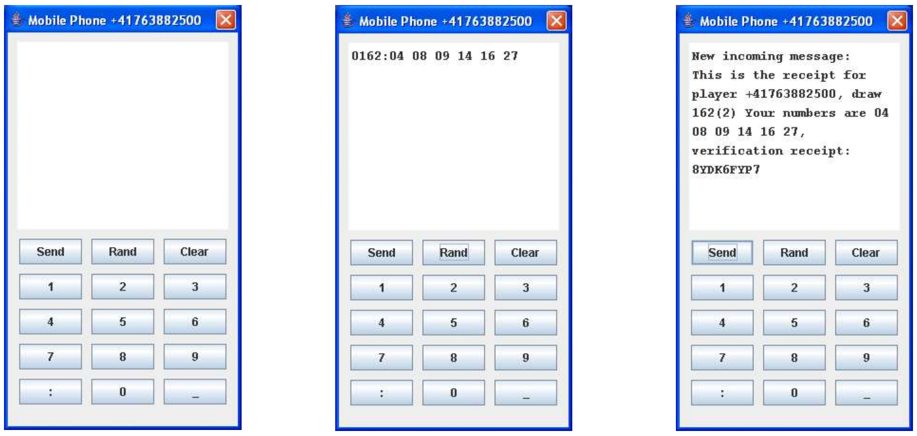
Where `DDDD` denotes a four-digit number of the draw and `NN NN NN NN NN NN` denotes the 6 chosen numbers. Note that every number has to be written with two digits, e.g. `04` instead of `4`.

By clicking on *Send*, the SMS is sent to the Lottery Organization (through the Service Provider). There it is checked and if it is accepted, a confirmation message will be shown in the *Playing Interface*. If it is not accepted, an error message will be shown which explains the reason for the refusal.



```
Tracer started...

[General] The Player was started with following arguments:
[General] identifier: +41763882500
[General] port: 8081
[General] service provider: localhost:8082
[General] lottery organization: localhost:8083
[General] public key is loaded from sms.pmova

[General] Try to connect to Service Provider.
[General] Try to connect to Lottery Organization.

[Protocol] START Protocol Subscribe
[Protocol] END Protocol Subscribe

[General] Try to read public key from file sms.pmova
[General] Have successfully read public key from file sms.pmova

[MOVA] N = 18598534292212959185933510772243673300877554632
[MOVA] s = 751294
[MOVA] y = 1
```

Figure 5.3: Tracer window



```
0162:04 08 09 14 16 27
```

```
New incoming message:
This is the receipt for
player +41763882500, draw
162(2) Your numbers are 04
08 09 14 16 27,
verification receipt:
8YDK6FYP7
```

(A) (B) (C)

Figure 5.4: Playing interface

The button *Rand* is an aid to quickly test the program: It generates an SMS with random playing numbers.

By clicking on *Clear*, the display is erased.

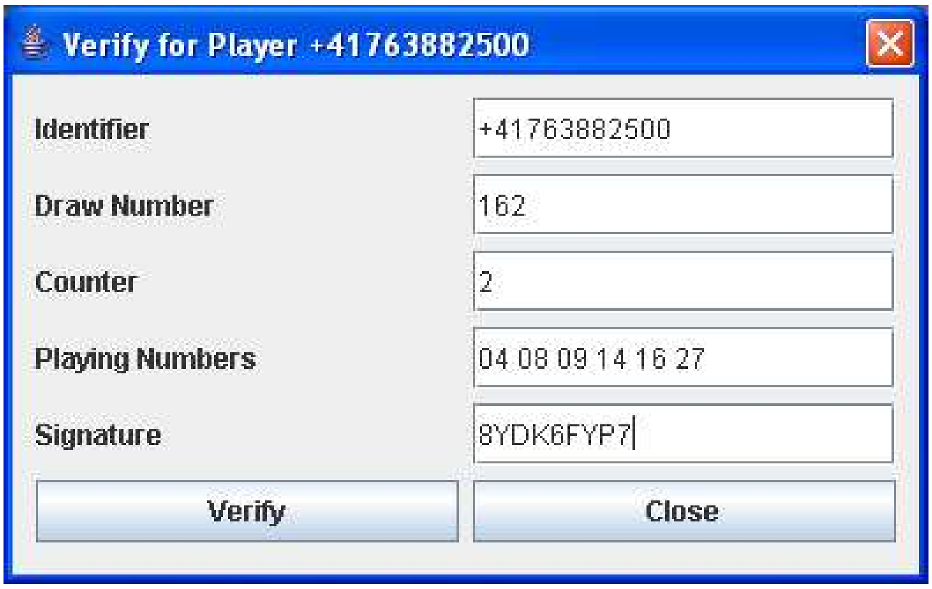
The text in the display can be changed either by using the keyboard of the computer, or by clicking with the mouse on the keys denoted with the numbers 0 to 9 and the characters : and a space denoted with "_".

Verification Interface

The window of the verification interface is shown in the Figure 5.5.

The player can verify the tickets by entering the corresponding data in the fields and clicking on *Verify*. He will then get one out of following replies:

- *The ticket is valid.* Due to the successful execution of the confirmation protocol, the player can be convinced that the signature is valid.
- *The ticket is not valid.* Due to the successful execution of the denial protocol, the player can be convinced that the signature is not valid.
- *This ticket is not registered.* A verification is not possible because it has never been played with this ticket.
- *The user has made too many attempts.* Every user has only access to the verification interface for a limited amount of attempts. If this limit is exceeded, the verification of tickets is not possible anymore.
- *Confirmation protocol abort.* The lottery organization claims that the ticket is valid, but it does not keep to the protocol specification of the



Identifier	+41763882500
Draw Number	162
Counter	2
Playing Numbers	04 08 09 14 16 27
Signature	8YDK6FYP7

Figure 5.5: Verification interface

confirmation protocol. If this happens in reality, there is either a technical problem in the IT infrastructure of the Lottery Organization, or the Lottery Organization tries indeed to cheat.

- *Denial protocol abort.* The lottery organization claims that the ticket is invalid, but it does not keep to the protocol specification of the denial protocol. Again, if this happens in reality, there is either a technical problem in the IT infrastructure of the lottery organization, or the lottery organization tries to cheat.

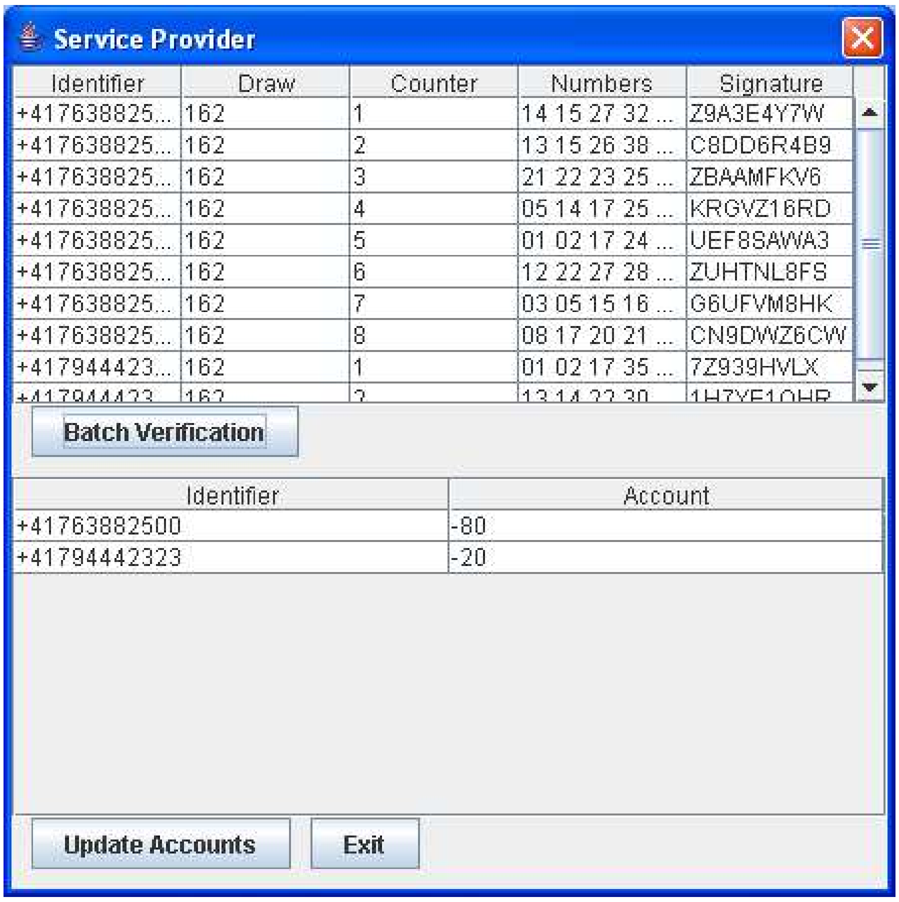
If the fields of the verification interface are not filled in a correct way, there will appear an error message.

Be aware that one reason for the messages *Confirmation protocol abort* and *Denial protocol abort* can be, that the Player uses a public key that does not match with the secret key of the lottery organization.

5.1.4 Using the Service Provider

When starting the Service Provider, two windows will appear. On the top there is the *Service Provider Control Window*, where the user can perform all the relevant actions. On the bottom the *Service Provider Tracer Window* appears, where the program shows in real time, what it is doing.

The *Service Provider Control Window* is shown in Figure 5.6.



Identifier	Draw	Counter	Numbers	Signature
+417638825...	162	1	14 15 27 32 ...	Z9A3E4Y7W
+417638825...	162	2	13 15 26 38 ...	C8DD6R4B9
+417638825...	162	3	21 22 23 25 ...	ZBAAMFKV6
+417638825...	162	4	05 14 17 25 ...	KRGVZ16RD
+417638825...	162	5	01 02 17 24 ...	UEF8SAWA3
+417638825...	162	6	12 22 27 28 ...	ZUHTNL8FS
+417638825...	162	7	03 05 15 16 ...	G6UFVM8HK
+417638825...	162	8	08 17 20 21 ...	CN9DWZ6CW
+417944423...	162	1	01 02 17 35 ...	7Z939HVLX
+417944423	162	2	13 14 22 30	1H7YF1QHR

Identifier	Account
+41763882500	-80
+41794442323	-20

Figure 5.6: Service Provider control window

Additionally, if the command-line option `-v` is set, a warning window will appear if the verification of the public key has failed.

The table on the top of the *Service Provider Control Window* shows all the tickets that have been accepted by the Lottery Organization. By clicking on *Batch Verification*, the batch verification protocol is executed for all the tickets that are stored in the table. If all the tickets are valid, the message *All the tickets are valid* appears. Otherwise, *Not all the tickets are valid* is shown.

The table on the bottom of the *Service Provider Control Window* show all the registered players together with the account balance. Whenever a player plays the lottery, the *ticket fee*[1] is withdrawn from his account. After the lottery organization has drawn the winning numbers, the user can enforce that the winnings[2] are credited to the players by clicking on *Update Accounts*.

5.1.5 Using the Lottery Organization

As the Service Provider, the Lottery Organization consists of two windows, the *Lottery Organization Control Window* and the *Lottery Organization Tracer Window*, which serves the same purpose than in the case of the Player and the Service Provider.

The *Lottery Organization Window* is shown in Figure 5.7

Additionally, if the command-line option `-v` is set, a warning window will appear if the verification of the public key has failed.

The checkbox *Behave honestly* defines whether incoming playing tickets are signed correctly. If it is not enabled, tickets are signed with a random string. If the checkbox *Accept playing* is enabled, incoming playing requests are accepted, otherwise they are rejected. The number in the field *Accepted draw* denotes the current draw, for which tickets are currently accepted.

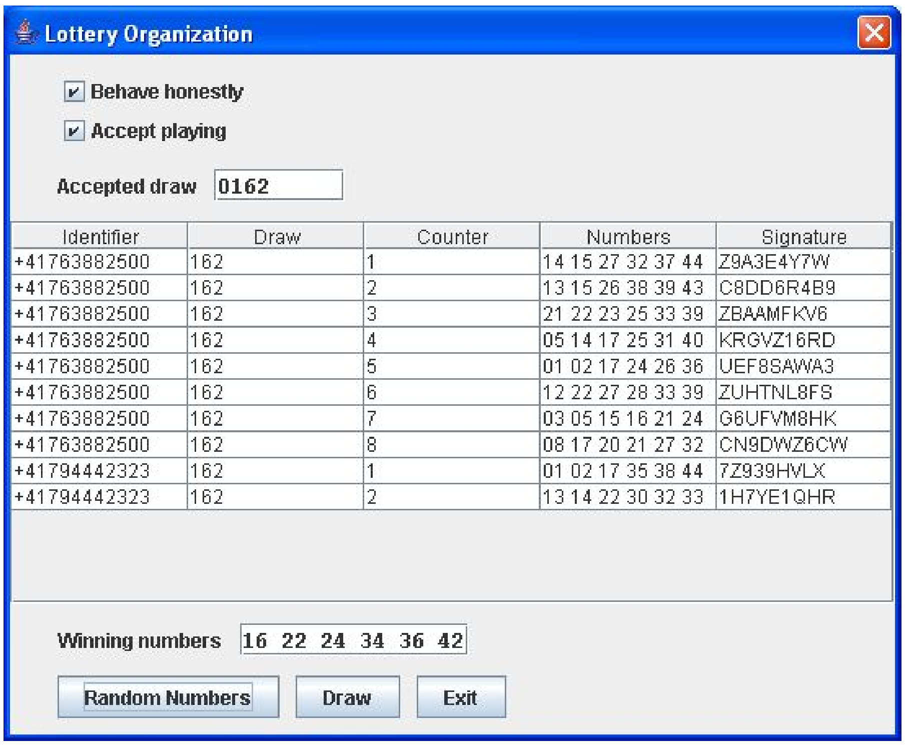
To summarize, a playing request is accepted, if the checkbox *Accept playing* is enabled and if the draw number corresponds to the number in the field *Accepted draw*.

The table in the middle of the *Lottery Organization Control Window* keeps track of the registered tickets. This is necessary to decide whether an incoming verification request will be accepted.

The user has the possibility to simulate the draw by entering the winning numbers in the field *Winning numbers*. To simplify the testing, a click on the button *Random Numbers* fills the field with random numbers. After clicking on *Draw*, the numbers are stored as the winning numbers. If the service provider clicks on *Update Accounts* afterward, these numbers are transfered to the service provider to determine the winnings of the players.

[1]As the program only serves as a demonstration for the feasibility, an imaginary price of 10 is taken as ticket fee.

[2]Also for the possible winnings, the program implements imaginary prices. For n correct numbers, the winning price is 10^{n-1} for $n \geq 3$ and 0 for $n < 3$.



✓ Behave honestly

✓ Accept playing

Accepted draw 0162

Identifier	Draw	Counter	Numbers	Signature
+41763882500	162	1	14 15 27 32 37 44	Z9A3E4Y7W
+41763882500	162	2	13 15 26 38 39 43	C8DD6R4B9
+41763882500	162	3	21 22 23 25 33 39	ZBAAMFKV6
+41763882500	162	4	05 14 17 25 31 40	KRGVZ16RD
+41763882500	162	5	01 02 17 24 26 36	UEF8SAWA3
+41763882500	162	6	12 22 27 28 33 39	ZUHTNL8FS
+41763882500	162	7	03 05 15 16 21 24	G6UFVM8HK
+41763882500	162	8	08 17 20 21 27 32	CN9DWZ6CW
+41794442323	162	1	01 02 17 35 38 44	7Z939HVLX
+41794442323	162	2	13 14 22 30 32 33	1H7YE1QHR

Winning numbers 16 22 24 34 36 42

Figure 5.7: Lottery Organization control window

5.2 The Java classes

Except the mathematical operations, where performance is an important issue, all the programming is done in Java. The key advantages of Java are platform independence, its powerful libraries and the possibility to easily create a user interface. Other reasons to choose Java were my familiarity with the language and the preexistence of some MOVA functionality in Java programmed by Sylvain Pasini, an assistant of the LASEC laboratory.

For the user interface we make use of the *SWING* library, which offers powerful functionality for designing the graphical representation as well as for the proceeding of user input. To represent big numbers, such as the prime numbers, we use the class *BigInteger*, which allows to store arbitrary-precision integers and the most important operations on them. For the communication between the different parts (Player, Service Provider and Lottery Organization) we use *Sockets*, which enable endpoint communication between different machines in a very simple way.

All the Java classes and most of their methods are commented according to the standard *Javadoc*. This allows the automated generation of HTML files which can be browsed to access the documentation in an easy way. These files can be found in the directory `SMSLotteryAppl/java/src/doc`.

All the Java classes are partitioned into two packages: `ch.epfl.lasec.mova` and `ch.epfl.lasec.sms_lottery`. The package `ch.epfl.lasec.mova` offers the full functionality of the MOVA signature scheme, such as key generation and batch verification. It is completely independent from the SMS Lottery Simulator and can be used by any other program that needs some functionality of the MOVA signature scheme. The package `ch.epfl.lasec.sms_lottery` on the other hand implements all the functionality of the SMS Lottery Simulator. The following listing gives a survey of the different classes of the two packages:

5.2.1 Package ch.epfl.lasec.mova

- `ByteStream` provides functionality to transfer special data objects, such as `String` objects and `BigInteger` objects over an `InputStream` and `OutputStream`.
- `ConfirmObj` is a data object containing data for the *Confirmation* protocol.
- `DenialObj` is a data object containing data for the *Denial* protocol.
- `KeyGeneration` generates a key pair.
- `MovaHelper` provides several methods related to the MOVA signature scheme, such as computing the Legendre symbol.
- `MovaPublicKey` contains the public key.
- `MovaSecretKey` contains the secret key.

- `Parameters` contains all the necessary parameters, such as key length, signature length, etc.
- `ProtocolBatchVerification` implements the protocol for the batch verification.
- `ProtocolConfirmation` implements the *Confirmation* protocol.
- `ProtocolDenial` implements the *Denial* protocol.
- `Tracer` is needed by most of the classes to give out tracing information in a tracer window.
- `TracerWindow` implements the graphical user interface of a tracer.

5.2.2 Package ch.epfl.lasec.sms_lottery

In Figure 5.8, we can see the main architecture of the SMS Lottery Simulator. From top to down, we have the classes of the Player, the Service Provider and the Lottery Organization. Every box corresponds to a class. A thick arrow between two classes means that the classes hold a reference to each other. Therefore, these objects are strongly tied together. A thin dotted line denotes that these two classes exchange data using sockets. All these data exchanges are described in the Section 5.4. The classes on the right side (written in italic style) represent the *graphical user interface* (GUI). Note that the Player maintains three different windows, the `PlayerWindow`, the `PlayerMobilePhone` and the `PlayerVerifyWindow`.

- `Helper` provides some auxiliary functionality for the SMS Lottery Simulator.
- `LotteryOrganization` is the main program for the Lottery Organization.
- `LotteryOrganizationDispatcher` handles incoming requests from `PlayerClient` and `ServiceProviderClient`.
- `LotteryOrganizationWindow` implements the graphical user interface of the Lottery Organization.
- `MessageWindow` shows a simple window with a text and an *okay* button.
- `Player` is the main program for the Player.
- `PlayerClient` communicates with the `ServiceProviderDispatcher` and the `LotteryOrganizationDispatcher` to execute the protocols.
- `PlayerMobilePhone` implements the graphical user interface of the Player that allows to play the lottery and receive SMS.
- `PlayerVerifyWindow` implements the graphical user interface of the Player that allows to verify the validity of a signature.

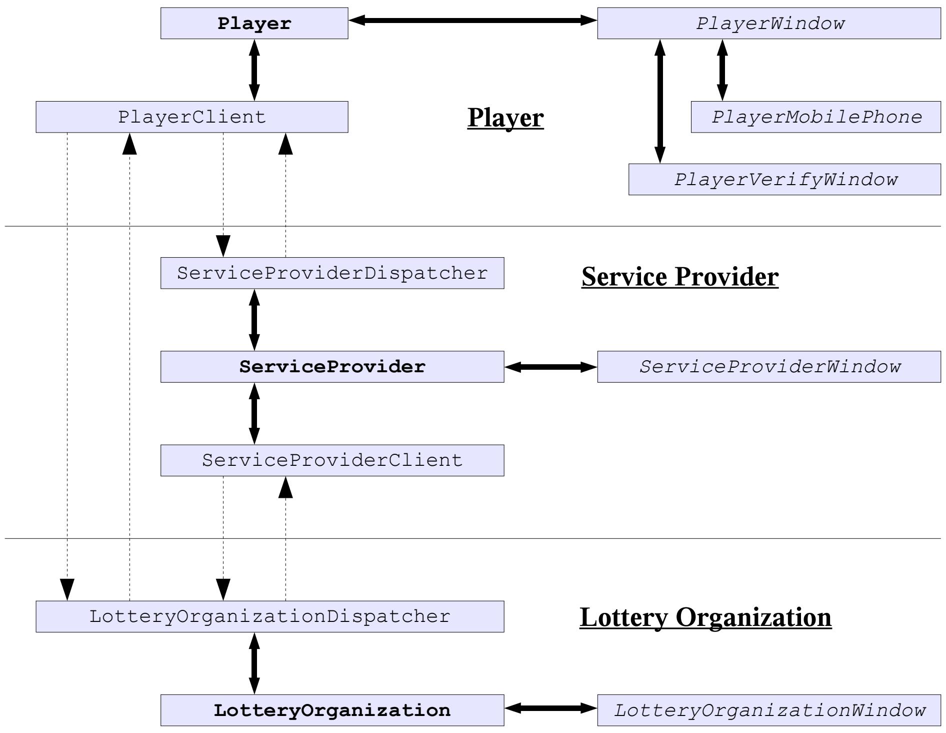


Figure 5.8: Architecture of SMS Lottery Simulator

- `PlayerWindow` implements the graphical user interface of the Player where a user can start the playing interface and the verification interface.
- `ServiceProvider` is the main program for the Service Provider.
- `ServiceProviderClient` communicates with the `LotteryOrganizationDispatcher` to execute the protocols.
- `ServiceProviderDispatcher` handles incoming requests from `PlayerClient` and `ServiceProviderClient`.
- `ServiceProviderWindow` implements the graphical user interface of the Service Provider.
- `Ticket` represents a ticket with the respective fields (*identifier*, *draw number*, etc.).

5.3 The C Programs

The performance-critical mathematical calculations are done by programs written in C. To execute these programs from Java, they are first compiled and then started like an external program, e.g. the following Java sequence starts the program *example*:

```
Runtime.getRuntime().exec("example" + parameters);
```

The C programs make use of the GNU multiple precision (GMP) library [GMP] to represent large numbers and to do basic calculations on them. To generate random numbers, they access the file `/dev/urandom` which is available in Linux and some Unix operating systems. It allows access to environmental noise collected from device drivers and other sources.

Following sub-programs are written in C and called from Java.

- `compute_one_u` computes u according to the formula $u = r^2 \cdot x_1^{d_1} \cdot x_2^{d_2} \prod_{k=1}^{\mathrm{Lsig}} {m_k}^{c_k} \mod N$. This is needed by the lottery organization in the confirmation protocol to verify whether the player has correctly computed his u_i.
- `compute_one_u_batch` works as `compute_one_u` but takes into consideration that we have multiple messages.
- `compute_u` computes a series of `ConfirmObj` objects. More precisely, it chooses $r_i \in_U \mathbb{Z}_n^*$ and $d_{i,1}, d_{i,2}, c_{i,1}, ..., c_{i,\mathrm{Lsig}} \in_U \mathbb{Z}_d$ for $i = 1, ..., \mathrm{Icon}$ and computes $u_i = r_i^2 \cdot x_1^{d_{i,1}} \cdot x_2^{d_{i,2}} \prod_{k=1}^{\mathrm{Lsig}} {m_k}^{c_{i,k}} \mod N$ for all i. These values are needed by the Player in the confirmation protocol.
- `compute_u_batch` works as `compute_u` but takes into consideration that we have multiple messages.

- `denial_compute_u` computes a series of `DenialObj` objects. More precisely, it chooses $r_i \in_U \mathbb{Z}_n^*, a_{i,1,1}, a_{i,2,1}, a_{i,1,2}, a_{i,2,2}, ..., a_{i,1,\mathrm{Lsig}}, a_{i,2,\mathrm{Lsig}} \in_U \{0,1\}, \lambda_i \in_U \{0,1\}$ and computes $u_{i,k}$ and $v_{i,k}$ for $i = 1, ..., \mathrm{Iden}$ and $k = 1, ..., \mathrm{Lsig}$. These values are needed for the Player in the denial protocol.
- `generate_keys`: generates key files without a validity proof.
- `generate_valid_keys`: generates key files where a validity proof is included in the public key file.
- `get_xi` applies a given seed on a pseudorandom generator to calculate some elements in G. This is needed to generate x_1 and x_2 from the public key. Furthermore it is needed for the calculation of the values that are defined by a message by taking the hash value from the message as the seed.
- `legendre` computes the Legendre symbols $\left(\frac{x_1}{p}\right), ..., \left(\frac{x_n}{p}\right)$ on a set of values $x_1, ..., x_n$.

5.4 The Protocols

As the SMS Lottery Simulator consists of different independent programs that can be run on different physical machines, there must be a mechanism to exchange data between them. As communication medium we have chosen *sockets*, which is supported by Java and offers bi-directional communication. In this section we present all the protocols that are used by the SMS Lottery Simulator to exchange data between the different programs. The only exceptions are the protocols *Confirmation* and *Denial*, which are described in Sections 5.5.1 and 5.5.2.

For the description of the protocols we use following notations:

- Sans serif (e.g. ok): Constants, that are literally transmitted
- Italic (e.g. *var*): Placeholders, that can take different values

We have three different classes of protocols:

- Protocols between the Player and the Service Provider
- Protocols between the Player and the Lottery Organization
- Protocols between the Service Provider and the Lottery Organization

5.4.1 Protocols between Player and Service Provider

Player Subscribe

This protocol is automatically executed when a new Player is started. It enforces the Service Provider to generate an account for the corresponding Player and to initialize it with 0.

1. Player → Service Provider: player subscribe.
2. Service Provider → Player: ok.
3. Player → Service Provider: *id*, where *id* is the identifier of the Player.
4. Service Provider → Player: ok.

Play

This protocol is executed every time when the player sends an SMS, e.g. clicks on the button *Send* on the virtual mobile phone.

1. Player → Service Provider: play.
2. Service Provider → Player: ok.
3. Player → Service Provider: *sms*, where *sms* is the text which is shown on the mobile phone when the button *Send* is clicked.
4. Service Provider → Player: ok.
5. Player → Service Provider: *id*, where *id* is the identifier of the Player.
6. Service Provider → Player: *sms_reply*, where *sms_reply* is the reply, which will be shown on the mobile phone as an incoming message.

5.4.2 Protocols between Player and Lottery Organization

Verification

This protocol is executed when the player accesses the verification interface, e.g. when he clicks on *Verify* in the Verify Window.

1. Player → Lottery Organization: verification.
2. Lottery Organization → Player: ok.
3. Player → Lottery Organization: *data*, where
4. Lottery Organization → Player: *response*, where *response* is one of the following numbers:
   - *response* = 1: The lottery organization claims that the signature is *valid*. In this case the Player executes a confirmation protocol with the lottery organization.
   - *response* = 2: The lottery organization claims that the signature is *invalid*. In this case the Player executes a denial protocol with the lottery organization.
   - *response* = 3: The lottery organization claims that the ticket is not registered and therefore refuses to prove the (in)validity.
   - *response* = 4: The lottery organization claims that there have been too many attempts and therefore refuses to prove the (in)validity.

5.4.3 Protocols between Service Provider and Lottery Organization

Service Provider Subscribe

This protocol is executed when the Service Provider is started. It is a simple handshake protocol to subscribe the Service Provider.

1. Service Provider → Lottery Organization: service provider subscribe.
2. Lottery Organization → Service Provider: ok.

Transfer SMS

Whenever the player plays the lottery, he sends an SMS to the service provider. The Service Provider thereafter makes use of the protocol *Transfer SMS* to send the content of the SMS to the lottery organization.

1. Service Provider → Lottery Organization: transfer sms.
2. Lottery Organization → Service Provider: ok.
3. Service Provider → Lottery Organization: *sms*, where *sms* is the content of the SMS that has been sent by the player.
4. Lottery Organization → Service Provider: ok.
5. Service Provider → Lottery Organization: *id*, where *id* is the identifier of the corresponding player.
6. Lottery Organization → Service Provider: *reply*.

Depending on the reply of the Service Provider, we have two possibilities:

If *reply* = ac, the lottery organization accepts the playing ticket.

1. Service Provider → Lottery Organization: ok.
2. Lottery Organization → Service Provider: *draw*, *counter_number*, *playing_numbers*, *signature* where these variables have their self-explanatory meaning.

If *reply* = nac, the lottery organization does not accept the playing ticket.

1. Service Provider → Lottery Organization: ok.
2. Lottery Organization → Service Provider: *errCode*, where *errCode* can have one of the following values:
   - the SMS does not have the right format, as there is no ":" at position 5 of the SMS
   - the first 4 characters do not encode numbers
   - the playing numbers do not match the required format

- the playing numbers are not in the right range (e.g. between 1 and 45)
- the number of the currently open draw does not match with the number of the requested draw
- no tickets are currently accepted

Get Draw

When a draw is simulated, the winning numbers are only stored at the lottery organization. To update the account data, the service provider has to obtain these numbers including the current draw number. It obtains the current draw number by executing the protocol *Get Draw*:

1. Service Provider $\rightarrow$ Lottery Organization: get draw.
2. Lottery Organization $\rightarrow$ Service Provider: $drawNr$, where $drawNr$ is the number of the current draw.

Get Winning Numbers

By executing this protocol, the service provider obtains the current winning numbers.

1. Service Provider $\rightarrow$ Lottery Organization: get winning numbers.
2. Lottery Organization $\rightarrow$ Service Provider: $winning\_numbers$, where $winning\_numbers$ are the current winning numbers.

5.5 Closer Look at some Program Parts

5.5.1 Confirmation Protocol

Note that in this section as well as in Section 5.5.2, we consider by Hom() the following function: $\mathrm{Hom}(x) := \log_{-1}(\left(\frac{x}{p}\right))$ where $\log_{-1}$ maps 1 to 0 and -1 to 1, and p is the prime number which is part of the secret key.

Given the confirmation protocol of Section 3.2 and our choices of Section 4.5, our confirmation protocol consists of following 5 steps:

1. For $i = 1, ..., \mathrm{Icon}$, the Player
   - chooses randomly $r_i \in_U \mathbb{Z}_n^*$ and $d_{i,1}, d_{i,2}, c_{i,1}, ..., c_{i,\mathrm{Lsig}} \in_U \{0,1\}$
   - computes $u_i = r_i^2 \cdot x_1^{d_{i,1}} \cdot x_2^{d_{i,2}} \prod_{k=1}^{\mathrm{Lsig}} {m_k}^{c_{i,k}} \mod N$ where $x_1, x_2 \in G$ are the elements from the public key and $m_k \in G$ for $k = 1, ..., \mathrm{Lsig}$ are the values that are specified by the message.

   All the u_i are now sent from the Player to the Lottery Organization.
2. The Lottery Organization computes for each u_i the homomorphism: $v_i = \mathrm{Hom}(u_i)$, commits to all u_i and sends the commitment to the Player.

3. The Player reveals now the values that he has randomly chosen to the Lottery Organization: He sends all $r_i, d_{i,1}, d_{i,2}, c_{i,1}, ..., c_{i,\mathrm{Lsig}}$ to the Lottery Organization

4. The Lottery Organization verifies whether the u_i were computed correctly, and opens the commitment by sending the opening information to the Player.

5. The Player accepts the protocol if the commitment opening information is correct, and the v_i correspond to the values he computed by using the homomorphic property: $v_i = y_1^{d_{i,1}} \cdot y_2^{d_{i,2}} \prod_{k=1}^{\mathrm{Lsig}} sig_k{}^{c_{i,k}} \mod N$. Where sig_k correspond to the elements of G representing the signature.

The confirmation protocol is implemented in the class `ProtocolConfirmation` of the package `ch.epfl.lasec.mova`. This class offers two methods: one for the prover (the lottery organization in our case) and one for the verifier (the player). The method for the prover needs the public key *and* the secret key, as well as a *seed* from the message. The method for the verifier only needs the public key, the *seed* from the message and the *signature*. Beside that, both methods also need some information about the communication channels.

For the step 1, the verifier generates Icon instances of the object `ConfirmObj` and initializes them by calling the method `compute_u` of class `MovaHelper`, which chooses all the values randomly and calculates u. As this requires a lot of computations, it is delegated to the C program `compute_u`.

For the step 2, the Lottery Organization computes the Legendre symbol on the received values u_i by calling the method `legendre()` of the class `MovaHelper`, which again executes the C program `legendre`. To commit to these values, the Lottery Organization chooses a random value and computes a hash[3] value on all the u_i and the random value. The opening information is therefore the random value.

Step 3 is implemented straightforward. In step 4, the lottery organization makes use of the C program `compute_one_u` to verify whether the player has correctly computed the u_i. Step 5 is again implemented straightforward.

5.5.2 Denial Protocol

Given the denial protocol of Section 3.2 and our choices of Section 4.5, the denial protocol of our lottery system consists of following 5 steps:

1. For $i = 1, ..., \mathrm{Iden}$, the Player
   - chooses randomly $r_i \in_U \mathbb{Z}_n^*$ and $a_{i,1,1}, a_{i,2,1}, a_{i,1,2}, a_{i,2,2}, ..., a_{i,1,\mathrm{Lsig}}, a_{i,2,\mathrm{Lsig}} \in_U \{0,1\}$ and $\lambda_i \in_U \{0,1\}$
   - computes
     $u_{i,1} = r_{i,1}^2 \cdot x_1^{a_{i,1,1}} \cdot x_2^{a_{i,2,1}} \cdot \lambda_i m_1 \mod N$
     $u_{i,2} = r_{i,2}^2 \cdot x_1^{a_{i,1,2}} \cdot x_2^{a_{i,2,2}} \cdot \lambda_i m_2 \mod N$

[3]In the SMS Lottery Simulator we use *SHA-1* as hash algorithm

...
$u_{i,\mathrm{Lsig}} = r_{i,\mathrm{Lsig}}^2 \cdot x_1^{a_{i,1,\mathrm{Lsig}}} \cdot x_2^{a_{i,2,\mathrm{Lsig}}} \cdot \lambda_i m_{\mathrm{Lsig}} \mod N$
where $x_1, x_2 \in G$ are the elements from the public key and $m_k \in G$ for $k = 1, ..., \mathrm{Lsig}$ are the values that are specified by the message.
Furthermore he computes
$w_{i,1} = y_1^{a_{i,1,1}} \cdot y_2^{a_{i,2,1}} \cdot \lambda_i sig_1 \mod 2$
$w_{i,2} = y_1^{a_{i,1,2}} \cdot y_2^{a_{i,2,2}} \cdot \lambda_i sig_2 \mod 2$
...
$w_{i,\mathrm{Lsig}} = y_1^{a_{i,1,\mathrm{Lsig}}} \cdot y_2^{a_{i,2,\mathrm{Lsig}}} \cdot \lambda_i sig_{\mathrm{Lsig}} \mod 2$
where $y_1, y_2 \in \{0, 1\}$ are the according elements from the public key (character of degree 2 of x_1 and x_2) and sig_k correspond to the elements of G representing the signature.

The player sends all $u_{i,k}$ and $v_{i,k}$ for all $i = 1, ..., \mathrm{Iden}$ and $k = 1, ..., \mathrm{Lsig}$.

2. The lottery organization finds the element of the signature that does not correspond to its homomorphism and finds the values of the λ_i by computing the homomorphism of all $u_{i,k}$ for $i = 1, ..., \mathrm{Iden}$ and $k = 1, ..., \mathrm{Lsig}$. It then commits to the values of λ_i and sends the commitment to the player.

3. The Player reveals now the values that he has randomly chosen to the Lottery Organization: He sends all $r_i, a_{i,1,1}, a_{i,2,1}, a_{i,1,2}, a_{i,2,2}, ..., a_{i,1,\mathrm{Lsig}}, a_{i,2,\mathrm{Lsig}}, \lambda_i$ for $i = 1, ..., \mathrm{Iden}$ to the Lottery Organization.

4. The Lottery Organization verifies whether the u_i and w_i were computed correctly, and opens the commitment by sending the opening information to the Player.

5. The Player accepts the protocol if the commitment opening information and the λ_i from the lottery organization are correct.

The denial protocol is implemented in the class `ProtocolDenial` of the package `ch.epfl.lasec.mova`. This class offers two methods: one for the prover (the lottery organization) and one for the verifier (the player). The method for the prover needs the public key *and* the secret key, as well as a *seed* from the message and the signature for which he wants to prove the invalidity. The method for the verifier only needs the public key, the *seed* from the message and the *signature*. Beside that, both methods also need some information about the communication channels.

For step 1, the verifier generates Iden instances of the object `DenialObj` and initializes them by calling the method `denial_compute_u` of class `MovaHelper`, which choosing all the values randomly and calculates u_i and v_i. As this requires a lot of computations, it is delegated to the C program `denial_compute_u`.

For the step 2, the Lottery Organization computes the Legendre symbol on the received values u_i by calling the method `legendre()` of the class `MovaHelper`, which again executes the C program `legendre`. After having computed λ_i, it commits to these values by chooses a random value and computing a hash value on all these λ_i and the random value. The opening information is the random value.

Steps 3 to 5 are straightforward.

Chapter 6

Conclusions and Outlook

With this work we have first of all shown that the MOVA signature scheme is perfectly applicable for the SMS lottery application. We have identified the main options for the implementation and weighted their advantages and disadvantages. Taking into account these options, we have proposed an SMS lottery system, which is in our opinion suitable to be implemented in practice. This lottery system specifies the roles of the players, the service provider and the lottery, as well as the interaction between these parties. We have elaborated a security model, which allowed us to prove that a signature size of 40 bits is sufficient. By implementing the system into a Java application we have proved that it is technically perfectly feasible.

Despite the technical feasibility and security, a real-world realization of our system depends on many other factors, such as:

- Would players understand how the security works in a system with undeniable signatures?
- Would players trust such a system?
- Could a realization of our system increase the playing volume?
- What are the costs of the implementation?

Answers to this questions require deeper business analysis. This could be subject of further work.

Appendix A

Case-Study: Swiss Lottery

The *SWISSLOS Interkantonale Landeslotterie* is an association in the possession of the 19 cantons of the German part of Switzerland and the canton of Ticino[1]. SWISSLOS offers a variety of lottery games, such as *Swiss Lotto* and *Euro Millions.* The net win from these products goes to 100% into public utility, to support social projects and projects for culture, nature and national sports, such as the Swiss Football Association and Swiss Olympic. In 2004 the gross yield out of playing, which is the total amount of playing fees minus the total amount of outpaid winnings, was 460 Mio CHF.

The players have to choose 6 distinct numbers out of 45. At the draw, 6 *normal* numbers and an *additional* number are dedicated. The order of the drawn numbers is irrelevant. Every player can win by following possibilities:

- rank 1: six numbers are right
- rank 2: five numbers *and* the *additional* number are right
- rank 3: five numbers are right
- rank 4: four numbers are right
- rank 5: three numbers are right

If there is only one winner for one rank, then this winner gets all the money of this rank. If there are two or more winners for one rank, the winning money of this rank is distributed in equal shares. If the shares for a higher rank are lower then for a lower rank, then these two ranks are put together and all the money is distributed to all the players of both ranks.

Every player in the rank 5 gets CHF 6 and every player in the rank 4 gets CHF 50.

From the net winning sum following amounts are removed:

- 10 percent for the rank 2 (if there is at least one winner in this category)
- the corresponding amount for the ranks 4 and 5

[1]The Loterie Romande is in charge of the lottery in the French-speaking part of Switzerland.

The remaining amount is equally distributed for the ranks 1 and 3.

If after application of these rules, a winner of rank 3 would win less than CHF 50, then the money for the rank 5 is reduced to CHF 5.

If nobody is in rank 1, the winning money (except CHF 100,000) of rank 1 is added to the winning money of rank 1 of the next draw. The CHF 100,000 are added to the rank 2.

Table A.1 shows the number of winnings of the possible classes and the average winning money of these classes for the year 2005 (for a total of 105 draws).

Winning class	Number of winnings	Average winning money
Rank 1	39	CHF $2,056,236.25$
Rank 2	279	CHF $111,781.20$
Rank 3	11’114	CHF $7,879.24$
Rank 4	530’787	CHF 50
Rank 5	8’356’947	CHF 6

Table A.1: Winning statistics for 2005

In 2005, there was a total price money of CHF 275,631,124.30.

Small winnings up to CHF 50 are not susceptible to the value added tax. They can be encashed at any point of sale for lottery tickets. Winnings between CHF 50 and CHF 1,000 can either be encashed at any point of sale (if they have enough money) or through *SWISSLOS* by sending the winning ticket to them.

Appendix B

Technical Data of SMS

SMS stands for *Short Message Service* and is a service to transmit short text messages from one mobile phone to another. The word SMS is not only used for the service, but also for the text message itself. When the SMS was brought onto the market in 1992, nobody has expected that this service will become so popular. In 2003 there were more than 16 Billions of SMS sent per month in Europe.

Every SMS contains two parts: The *header* and the *body*. The header contains context information such as the telephone numbers of the addresser and receiver, the character encoding or the character set. The size of the body is limited to 1,120 bit. For the character encoding there are three possibilities:

- 7 Bit: for Latin letters, (capital and small), Greek letters (only capital) and some special characters. One SMS can contain up to 160 characters (160 characters $\times$ 7 bit/character = 1,120 bit)
- 8 Bit: for binary data. One SMS can contain up to 140 bytes (140 bytes = 1,120 bit)
- 16 Bit: for *Unicode* encoding (UTF-16) for 2-byte characters in languages such as Arabic or Chinese. One SMS can contain up to 70 characters (70 characters $\times$ 16 bit/character = 1,120 bit)

To send longer content, one can use a series of SMS (known as *long SMS* or *concatenated SMS*). In this case, the message is segmented over multiple messages, which all have to start with a *user data header*. Due to this additional header, the payload per segment is reduced in the following way:

- 7 Bit: 153 characters
- 8 Bit: 134 characters
- 16 Bit: 67 characters

The standard theoretically allows up to 255 segments, 3 to 4 segments are the practical maximum.

In the western world, 7 Bit encoding without segmentation is the mostly used format. In this format we can use 160 characters.

Appendix C

SMS Lotteries that are Used in Practice

There are several SMS lotteries used in practice, as for example:

- *LottoFun SMS* from the *South African National Lottery* [1]
- *FirstLottoSMS* from the *At-firstcall GmbH* with access to the *German National Lottery* [2]
- *LottoSMS* in Belgium [3]
- *UK National Lottery* [4]

None of them is using cryptographic methods to guarantee the validity of a ticket.

Appendix D

Security Reduction for Universal Forgery

In this section we give a reduction from the *universal forgery* game with a *chosen-message attack* to the *GHI*-Problem for the MOVA Signature Scheme. It has been developed during the elaboration of the security model for the SMS Lottery. Even if it is not used anymore in the final version of the security model, it is a noticeable result, and therefore we put it in the annex.

The bound of our reduction from the *universal forgery* game is tighter than the one of [M06] for the *existential forgery* game. Therefore it is useful to estimate the signature length for applications that only need to be secure against *universal forgery*, but not against *existential forgery*. The *universal forgery* game is defined as follows:

Definition 18. Game^{uf-cma}*: The forger* $\mathcal{F}$ *only knows the public key and receives a message* m *that is uniformly at random chosen from the message space. She can then access the* GenS *oracle, the* Sign *oracle for any message* $m' \neq m$ *and the* Verify *oracle for any pair* (m', sig)*. She has to output a signature for* m *and wins, if the signature is valid.*

Note that Definition 18 is specific to the MOVA signature scheme. To estimate the signature length, we can make use of following theorem:

Theorem 7. *Consider the Lsig-S-GHI problem with the same parameters as for the MOVA scheme, i.e.,* $G = \text{Xgroup}, H = \text{Ygroup}$*. Assume that for any solver* $\mathcal{B}$ *with a given complexity we have*

$$\mathsf{Succ}_{\mathcal{B}}^{Lsig-S-GHI} \leq \varepsilon$$

Then, any forger $\mathcal{F}$ *with similar complexity, which tries to forge a signature for* m*, and can make queries to* GenS*, queries to a signing oracle* Sign *(for any message other than* m*) and* q_v *queries to the confirmation/denial oracle* Verify*, wins the universal forgery game under adaptive chosen-message attack with a probability*

$$\mathsf{Succ}_{\mathcal{F}}^{uf-cma} \leq (q_v + 1)\varepsilon.$$

Proof. Let $\mathcal{F}$ be a forger who succeeds to forge the signature of a given message under an adaptive chosen-message attack with a non-negligable probability ε. We will construct an algorithm $\mathcal{B}$ which solves the Lsig-S-GHI problem with

$$S := \{(\mathrm{Xkey}_1, \mathrm{Ykey}_1), ..., (\mathrm{Xkey}_{\mathrm{Lkey}}, \mathrm{Ykey}_{\mathrm{Lkey}})\}$$

using the forger $\mathcal{F}$. At the beginning, $\mathcal{B}$ receives the challenges $x_1, ..., x_{\mathrm{Lsig}} \in$ Xgroup of the Lsig-S-GHI problem. Then $\mathcal{B}$ runs the forger by giving her a randomly generated message m and simulates the queries to the random oracle GenS, to the signing oracle Sign and to the denial/confirmation oracle Verify. We can assume that all messages sent to Sign resp. Verify were previously queried to GenS (since the oracle Sign resp. Verify has to make such queries anyway). $\mathcal{B}$ simulates the oracles GenS, Sign and Verify as follows:

- GenS: If the previously randomly generated message m is queried, he answers with x_1, ..., x_{Lsig}. If a new message is queried, he picks $a_{i,j} \in_U \mathbb{Z}_d$ and $r_i \in_U$ Xgroup uniformly at random for $i = 1, ..., \mathrm{Lsig}$ and $j = 1, ..., \mathrm{Lkey}$. He answers then

  $\mathrm{Xsig}_i := dr_i + \sum_{j=1}^{\mathrm{Lkey}} a_{i,j}\mathrm{Xkey}_j$ for $i = 1, ..., \mathrm{Lsig}$.

  For each message, $\mathcal{B}$ keeps the coefficients $a_{i,j}$ and r_i in the memory.

- Sign: For a message $m' \neq m$, he looks up the corresponding coefficients in the memory and answers with $\mathrm{Ysig}_i := \sum_{j=1}^{\mathrm{Lkey}} a_{i,j}\mathrm{Ykey}_j$ for $i = 1, ..., \mathrm{Lsig}$. Note that by definition, the forger $\mathcal{F}$ is not allowed to query the randomly generated message m to the Sign oracle.

- Verify: For a message $m' \neq m$ and a signature sig, he looks up the corresponding coefficients in the memory and easily deduces whether the signature sig is valid. He then simulates the appropriate protocol. If $\mathcal{F}$ queries the message m together with a signature sig, it is obviously not easy to determine whether this signature is valid or not. Therefore we use the following technique: Let's denote $(m,\ sig_i)$ the ith query to Verify with the randomly generated message m, and $(m,\ sig_{q_v+1})$ denote the output of $\mathcal{F}$. $\mathcal{B}$ tries now to guess the index ℓ of the first valid signature to m by simply picking ℓ uniformly at random in $\{1, ..., q_v + 1\}$. The Verify oracle behaves now for message pairs $(m,\ sig_i)$ as follows:

  - ⋆ $i < l$: $\mathcal{B}$ assumes that the signature sig_i is not valid and simulates the appropriate protocol.

  - ⋆ $i = l$: $\mathcal{B}$ assumes that the signature sig_i is valid and outputs the signature, which consists of Ysig_i.

If the probability for $\mathcal{F}$ to output the correct signature to m is $\mathsf{Succ}_{\mathcal{F}}^{uf-cma} = \delta$, the probability for $\mathcal{B}$ to retrieve the correct answer is $\mathsf{Succ}_{\mathcal{B}}^{Lsig-S-GHI} = \frac{1}{q_v+1}\delta$.

Therefore, if there does not exist any algorithm of a given complexity that can solve the Lsig-S-GHI problem with probability higher than ε, there does not exist any forger of similar complexity with access to the Sign oracle and up to q_v queries to the confirmation/denial oracle with the message m, that wins the universal forgery game under adaptive chosen-message attack with a probability higher than $(q_v + 1)\varepsilon$ $\square$

We show the improved bound for some examples. Table D.1 shows the required signature length for *Existential Forgery* with a security probability of 2^{-30} and Table D.2 shows the required signature length for the *Universal Forgery* with the same security probability. The computations are based on the same assumptions that are made in [M06], e.g. we can adjust Xgroup such that

$$\mathsf{Succ}_{\mathcal{B}}^{\text{Lsig-}S\text{-GHI}} \approx d^{-\text{Lsig}}$$

.

q_s	q_v	Lsig $\cdot \log_2(d)$ bits
2^{10}	2^{10}	52
2^{10}	2^{20}	62
2^{20}	2^{10}	62
2^{20}	2^{20}	72

Table D.1: Signature size for existential forgery

q_s	q_v	Lsig $\cdot \log_2(d)$ bits
2^{10}	2^{10}	41
2^{10}	2^{20}	51
2^{20}	2^{10}	41
2^{20}	2^{20}	51

Table D.2: Signature size for universal forgery

Bibliography

[1] http://www.lottofun.co.za/sms/sms.asp

[2] http://www.first-lotto-sms.com

[3] http://www.belgische-lotto.com/new/lottosms.php

[4] http://www.national-lottery.co.uk

[BLS01] Dan Boneh, Ben Lynn and Hovav Shacham. Short Signatures from the Weil Pairing. In Advances in Cryptography - AsiaCrypt 2001, volume 2248 of Lecture Notes in Computer Science, pages 514 - 532. Springer-Verlag, 2001.

[CA89] D. Chaum and H. van Antwerpen. Undeniable signatures. In Advances in Cryptology - CRYPTO 1989, volume 435 of Lecture Notes in Computer Science, pages 212 - 216. Springer-Verlag, 1990.

[FS86] Amos Fiat and Adi Shamir. How to Prove Yourself: Practical Solutions to Identification and Signature Problems. In Advances in Cryptology CRYPTO 1986, volume 263 of Lecture Notes in Computer Science, pages 186 194. Springer-Verlag, 1987.

[GMP] http://swox.com/gmp

[MV04] Jean Monnerat and Serge Vaudenay. Generic Homomorphic Undeniable Signatures. In Advances in Cryptology AsiaCrypt 2004, volume 3329 of Lecture Notes in Computer Science, pages 354 - 371. Springer-Verlag, 2004.

[M06] Jean Monnerat. Short Undeniable Signatures: Design, Analysis, and Applications. PhD thesis, Swiss Federal Institute of Technology Lausanne, 2006.

[V06] Serge Vaudenay. A Classical Introduction to Cryptography: Applications for Communications Security. Springer-Verlag, 2006.