

# Secure and Verifiable Electronic Elections at NTNU

**Bent Kristoffer Rosvold Onshus**

Master of Science in Communication Technology

Submission date: June 2006

Supervisor: Stig Frode Mjølunes, ITEM

Co-supervisor: Kristian Gjølsteen, ITEM



# Problem Description

It is hard to design efficient and secure election schemes for moderate to large scale applications.

The thesis should give a brief introduction to the theory of electronic elections, and a thorough study of a certain electronic election scheme by Damgård, Jurik and Nielsen (DJN scheme).

The functionality and security requirements for elections at NTNU should be analyzed, and it should be verified that the DJN scheme can satisfy these requirements.

An implementation of the DJN scheme should be made and used to test the feasibility of the scheme for elections at NTNU. The focus should be on computational requirements for vote creation, tallying and verification for the various elections at NTNU.

Assignment given: 16. January 2006  
Supervisor: Stig Frode Mjølhusnes, ITEM



# Abstract

This thesis describes an electronic voting system based on Damgård, Jurik and Nielsen's generalization of Paillier's probabilistic public key system. A threshold variant of this homomorphic cryptosystem is used to provide universally verifiable elections, where zero-knowledge proofs are used for proving correctness of votes.

Using this cryptosystem, an electronic voting system that supports voting for 1 out of  $L$  candidates is described. Two types of encoding may be used to prove the validity of the votes. The number of proofs needed using normal encoding is linear in  $L$ , while the number of proofs needed using binary encoding is logarithmic in  $L$ . It is shown how to extend the system to allow casting a vote for  $t$  out of  $L$  candidates. This method may easily be used to carry out elections with weighted votes without any added complexity to the system.

The system is shown to satisfy the requirements for elections at The Norwegian University of Science and Technology (NTNU). A fully functional implementation of the electronic voting system as a distributed system, using Java Remote Method Invocation, is presented. The implementation is used to analyze the feasibility of using this voting system for future elections at NTNU. The implementation is tested using various keylengths and various election parameters. With a keylength of 1024 bits, the simulated time for verification of complex elections is small enough to be considered universally verifiable.



# Preface

This thesis is written as a part of my Master of Technology degree at the Norwegian University of Science and Technology (NTNU) during the spring semester of 2006.

I would now like to use this opportunity to thank:

- My supervisor at the Department of Telematics Kristian Gjøsteen for valuable guidance during the work with this thesis.
- Kristin K. H. for knocking it up another notch.



# Table of Contents

<b>Abstract</b>	<b>i</b>
<b>Preface</b>	<b>iii</b>
<b>List of Figures</b>	<b>xi</b>
<b>List of Tables</b>	<b>xiii</b>
<b>List of Listings</b>	<b>xv</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Background . . . . .	1
1.2 Problem to be addressed . . . . .	2
1.3 Scope . . . . .	2
1.4 Outline . . . . .	3
<b>2 Electronic Voting Systems</b>	<b>5</b>
2.1 Electronic Voting . . . . .	5
2.2 Requirements for Voting Systems . . . . .	6
2.3 Requirements for elections at NTNU . . . . .	9
2.3.1 Functional Requirements . . . . .	9
2.3.2 Security Requirements . . . . .	9
2.4 Existing Electronic Voting Schemes . . . . .	11

2.4.1	Homomorphic Cryptosystems . . . . .	12
2.4.2	Mix-Nets . . . . .	12
2.4.3	Blind Signatures . . . . .	13
2.4.4	Publicly Verifiable Secret Sharing . . . . .	14
2.5	Summary . . . . .	15
<b>3</b>	<b>The Cryptographic Voting System</b>	<b>17</b>
3.1	Introduction . . . . .	17
3.2	Damgård, Jurik and Nielsen's generalized variant of Paillier's public-key system . . . . .	18
3.2.1	Key Generation . . . . .	19
3.2.2	Encryption . . . . .	19
3.2.3	Decryption . . . . .	19
3.3	Zero Knowledge Proofs . . . . .	21
3.4	The Zero-Knowledge Protocols . . . . .	22
3.4.1	Encryption of 0 - protocol . . . . .	22
3.4.2	Protocol 1-out-of-2 $n^s$ 'th power . . . . .	24
3.4.3	Protocol 1-out-of- $L$ $n^s$ 'th power . . . . .	25
3.4.4	Protocol Multiplication-mod- $n^s$ . . . . .	26
3.4.5	Protocol for Equality of Discrete Logarithms . . . . .	27
3.5	Non-Interactive Zero-Knowledge Proofs . . . . .	28
3.5.1	Random Oracle . . . . .	28
3.5.2	Fiat-Shamir Heuristic . . . . .	29
3.5.3	The Non-Interactive Zero-Knowledge Proofs . . . . .	29
3.6	A Threshold Variant of the Cryptosystem . . . . .	30
3.6.1	Key Generation . . . . .	31
3.6.2	Encryption . . . . .	32
3.6.3	Share Decryption . . . . .	32

3.6.4	Share Combination . . . . .	32
3.7	Summary . . . . .	32
<b>4</b>	<b>Design</b>	<b>35</b>
4.1	Top Level Architecture . . . . .	35
4.2	Parameters of the Election . . . . .	37
4.3	Model of Elections . . . . .	37
4.4	The Different Parts of the Voting System . . . . .	38
4.4.1	The Voter Application . . . . .	38
4.4.2	The Bulletin Board . . . . .	39
4.4.3	The Decryption Server . . . . .	39
4.4.4	The Election Authority . . . . .	40
4.5	Voting Methods . . . . .	40
4.5.1	Yes/No Election . . . . .	41
4.5.2	1-out-of- $L$ Election . . . . .	42
4.5.3	$t$ -out-of- $L$ Election . . . . .	43
4.5.4	1-out-of- $L$ Election with Binary Encoding of Votes . . . . .	45
4.5.5	$t$ -out-of- $L$ Election with Binary Encoding of Votes . . . . .	46
4.6	Satisfying Election Requirements at NTNU . . . . .	47
4.6.1	Functional Requirements . . . . .	47
4.6.2	Security Requirements . . . . .	47
4.7	Other Considerations . . . . .	49
4.8	Summary . . . . .	50
<b>5</b>	<b>Implementation</b>	<b>51</b>
5.1	Overview . . . . .	51
5.2	Implementation of the Cryptosystem . . . . .	52
5.3	Implementation of the Common Library . . . . .	54

5.4	Implementation of the Election Authority . . . . .	56
5.4.1	Description of the Structure . . . . .	56
5.4.2	Parameters Chosen by the Election Authority . . . . .	59
5.5	Implementation of the Voter Application . . . . .	59
5.5.1	Description of the Structure . . . . .	59
5.5.2	Optimizations of Encryption . . . . .	62
5.6	Implementation of the Decryption Server . . . . .	63
5.6.1	Description of the Structure . . . . .	64
5.6.2	Optimizations of the Decryption Server . . . . .	66
5.7	Implementation of the Bulletin Board . . . . .	66
5.8	The Simulation Tool . . . . .	68
5.9	Executing the Voting System . . . . .	69
5.10	Summary . . . . .	71
<b>6</b>	<b>Performance Evaluations</b>	<b>73</b>
6.1	Comparison of Voting Methods . . . . .	73
6.1.1	Parameters for the Performance Evaluation . . . . .	73
6.1.2	Performance Evaluation of Share Decryption and Vote Val- idation . . . . .	74
6.1.3	Performance Evaluation of Vote Creation . . . . .	75
6.1.4	Performance of the Election Authority . . . . .	76
6.2	Elections with Higher Security Requirements . . . . .	77
6.3	Feasibility Analysis . . . . .	78
6.4	Programming Language Comparison . . . . .	79
6.5	Summary . . . . .	79
<b>7</b>	<b>Concluding Remarks</b>	<b>81</b>
7.1	Conclusion . . . . .	81
7.2	Future Work . . . . .	82

<b>Bibliography</b>	<b>85</b>
<b>A Source Code</b>	<b>89</b>
A.1 Voter Application . . . . .	89
A.1.1 Precomputation of Values for Encryption . . . . .	89
A.1.2 Optimized Encryption Algorithm . . . . .	90
A.1.3 Creation of a <i>Vote</i> in a 1-out-of- $L$ Election . . . . .	90
A.1.4 Creation of a <i>MultipleVote</i> in a $t$ -out-of- $L$ Election . . . . .	92
A.1.5 Creation of a <i>BinaryVote</i> in a 1-out-of- $L$ Election . . . . .	94
A.1.6 Creation of a <i>MultipleBinaryVote</i> in a $t$ -out-of- $L$ Election . . . . .	98
A.2 The Share Decryption of Votes . . . . .	100
A.3 The Verification of the Zero-Knowledge Proofs . . . . .	101
A.3.1 Verification of a <i>Vote</i> in a 1-out-of- $L$ Election . . . . .	101
A.3.2 Verification of <i>MultipleVotes</i> in a $t$ -out-of- $L$ Election . . . . .	102
A.3.3 Verification of a <i>BinaryVote</i> in a 1-out-of- $L$ Election . . . . .	104
A.3.4 Verification of <i>MultipleBinaryVotes</i> in a $t$ -out-of- $L$ Election . . . . .	106
A.3.5 Verification of a <i>DecryptedShare</i> Acquired from a Decryp- tion Server . . . . .	108
A.4 The Combination of <i>DecryptedShares</i> Acquired from Decryption Servers . . . . .	109
<b>B Digital Material following the Thesis</b>	<b>113</b>



# List of Figures

4.1	Top level architecture of the voting system. . . . .	36
4.2	Possible layered structure of the system . . . . .	37
5.1	Deployment diagram of the voting system . . . . .	53
5.2	Class diagram of the package <i>votingSystem</i> . . . . .	55
5.3	Class diagram of the <i>ea</i> package . . . . .	57
5.4	The sequence diagram for the election authority . . . . .	58
5.5	Screenshot of the administration window at the election authority.	60
5.6	Class diagram of the package <i>voter</i> . . . . .	61
5.7	Sequence diagram for the voter application. . . . .	62
5.8	Class diagram of the <i>ds</i> package . . . . .	64
5.9	Sequence diagram for the actions of the decryption server. . . . .	65
5.10	Class diagram for the package <i>bb</i> containing the bulletin board. . .	67
5.11	Screenshot of the simulation results . . . . .	69



# List of Tables

2.1	Requirements for a voting system for elections at NTNU. . . . .	11
4.1	The parameters of the voting system. . . . .	37
4.2	Requirements satisfied by the voting system proposed in this thesis.	49
5.1	Package names and main classes for the components in the voting system. . . . .	70
6.1	The parameters chosen for the performance testing. . . . .	74
6.2	The time spent for validation of votes and vote multiplication per vote for different values of $L$ . . . . .	75
6.3	A comparison of mean share decryption time per vote in 2-out-of- $L$ elections using the two different encodings. . . . .	75
6.4	A comparison of mean vote encryption and proof creation time in 1-out-of- $L$ elections using the two different encodings. . . . .	76
6.5	A comparison of the two different encodings with respect to mean vote encryption and proof creation time in 2-out-of- $L$ elections. . .	76
6.6	A comparison of share combination time in all four voting methods.	77
6.7	A comparison of mean share decryption and vote validation times per vote using a 2048-bits key. . . . .	77



# List of Listings

3.1	Algorithm used for extracting $i \bmod n^{s+1}$ from $(1 + n)^i \bmod n^{s+1}$	20
5.1	Unoptimized encryption algorithm	62
5.2	Calculation of the precomputed values	63
5.3	Calculation of $(1 + n)^m$	63
A.1	Precomputation of values for encryption	89
A.2	Full optimized encryption algorithm	90
A.3	Creation of a <i>Vote</i> in a 1-out-of- $L$ Election	90
A.4	Creation of a <i>MultipleVote</i> in a $t$ -out-of- $L$ Election	92
A.5	Creation of a <i>BinaryVote</i> in a 1-out-of- $L$ election	94
A.6	Creation of a <i>MultipleBinaryVote</i> in a $t$ -out-of- $L$ election	98
A.7	Share decryption of votes and creation of a zero-knowledge proof for the calculations	100
A.8	Verification of a <i>Vote</i>	101
A.9	Verification of <i>MultipleVotes</i>	102
A.10	Verification of a <i>BinaryVote</i>	104
A.11	Verification of <i>MultipleBinaryVotes</i>	106
A.12	Verification of a <i>DecryptedShare</i>	108
A.13	Combination of <i>DecryptedShares</i> to a result	109



# Chapter 1

## Introduction

### 1.1 Background

Electronic voting systems that allow voting over Internet are starting to gain popularity around the world. Internet voting systems give the voter the possibility of voting from any Internet-accessible computer at any time of day. These systems are suitable for elections where voters are not needed to cast their votes in designated polling places. Elections in organizations and at universities are examples of elections which have requirements regarding security without requiring voters to go to polling places to vote.

The efficiency, flexibility and accessibility of Internet voting systems is attractive, but many systems used today lack important security properties. Up until today most internet voting systems have given administrators the possibility of viewing the votes given, often also checking the identity of voters. This has raised privacy issues, because the voters' choices are revealed. The correctness of the results can also be questioned because the systems can be vulnerable to election fraud from the inside, and voters are rarely able to verify the results. These problems have called for a secure electronic voting system that can provide a higher security against election fraud, and that can provide privacy, correctness and verifiability.

## 1.2 Problem to be addressed

The main focus of this thesis is to design and implement a secure electronic voting system suitable for elections at the Norwegian University of Science and Technology (NTNU). The system will provide a voting service that relies on cryptography to efficiently and securely hide the voters' choices from other participants in the election, as well as from people inside the electoral apparatus. The voting system will provide universally verifiable elections where anyone participating in the election can verify all votes cast, as well as the actions of the components in the voting system.

This thesis will describe the building blocks needed to create such a voting system. This will start by presenting the homomorphic cryptosystem used as a basis for the voting system. This cryptosystem will be used together with different zero-knowledge proofs to prove the validity of votes without revealing anything beyond the validity of the proof. These building blocks will then be combined into a fully functional voting system.

This thesis will also perform an analysis of the computational requirements for carrying out elections using this system. This analysis will be done both mathematically and by testing the actual system. Different encodings of votes and different parameters will be used to find the optimal performance of the system in different scenarios. The analysis will then be used to determine the feasibility of using this system in elections at NTNU.

## 1.3 Scope

This thesis will focus mainly on the voting system itself. The security of a voting system depends on many outside factors. Many problems exist concerning the deployment of a system like this in a real life situation. Examples include the key distribution, securing communication between components in the system and controlling access to the voting system. These are problems where adequate solutions exist today. Solving these problems for deployment of this system in an Internet environment is outside the scope for this thesis.

## 1.4 Outline

The following chapter will give the reader an introduction to electronic voting, where different properties are defined that may be used to characterize a voting system. These properties will be explained, along with solutions for satisfying the different properties. Four different types of voting schemes are then presented, along with an explanation of how they work, and how they satisfy the different properties.

Chapter 3 explains the homomorphic cryptosystem as well as zero-knowledge protocols to show that a given ciphertext encrypts one of a set of given plaintexts, and protocols to verify multiplicative relations of plaintexts. Then we explain how to transform these interactive protocols into non-interactive proofs using the Fiat-Shamir heuristic. Finally, the threshold variant of the cryptosystem is explained, which will be used by decryption server to collectively decrypt a given ciphertext.

Chapter 4 covers the design of the voting system. This is where the building blocks explained in Chapter 3 are combined to a fully functional voting system. All the components of the system are explained, along with the different voting methods available in the system. It is then shown how this voting system satisfies the requirements for electronic voting at NTNU.

Chapter 5 describes the implementation of this voting system in Java. The functionality of all components is thoroughly explained. Some optimizations are introduced to increase the performance of the system. This chapter also includes a tutorial for running the implementation of the system.

Chapter 6 gives a performance evaluation of the system with realistic parameters for NTNU elections, as well as some tests of elections with higher security demands. The different encodings of votes are analyzed, and the optimal encoding for different numbers of candidates is determined.

Chapter 7 covers the concluding remarks for this thesis, as well as some notes about future research areas that may be investigated.



## **Chapter 2**

# **Electronic Voting Systems**

This chapter will describe the general aspects of voting systems. Some properties that can be used for characterizing a voting system are defined, along with possible methods that can be used for satisfying these properties. Some existing voting schemes are explained along with an explanation of how these schemes satisfy the properties.

### **2.1 Electronic Voting**

A voting system is a means of choosing between a number of options, based on the input of a number of voters. A voting system consists of the rules for how voters express their desires, and how these desires are aggregated to yield a final result. Voting is perhaps best known to be used in elections where political candidates are selected for public office. It can also be used for any other situation where people are given the opportunity to choose between different alternatives, for instance price awards or different plans of action for organizations, businesses or even countries.

Electronic voting usually refers to collection and dissemination of peoples opinions, with the help of some machinery that is more or less computer supported. This concept is not new, and methods to record and count votes using electromechanical machinery was first proposed by Thomas Edison in 1869 [5]. Newer examples of electronic voting systems are punch-card systems that have been in use since the 1960s, and touch screen systems that are widely used today. These

newer electronic voting systems have the possibility of notifying the voter in case of invalid votes, as well as providing instant counts after polling.

Electronic voting systems can be realized in a variety of ways. For political elections, voting is often carried out using voting machines in designated polling places. For elections that do not require voters to go to a specific place to cast their votes, Internet voting may be used. Many elections held at universities or in organizations may benefit from the efficiency of Internet elections.

The voting system used for elections at NTNU today is a simple web form secured by SSL. Authentication is performed using the normal credentials used to access the internal network at NTNU. A voter's choice is visible to the people inside the electoral apparatus, and there is no guarantee against electoral fraud from the inside. The actions of the voting system is not verifiable for the participants in the election, and the system is more or less a black box where votes go in and a result is shown at the end of the election. Voters have no way of verifying that the result is correct and that the votes cast are unmodified.

## 2.2 Requirements for Voting Systems

Voting systems may have varying requirements. When viewers cast votes for participants in a television reality show, revealing the identity of voters and election fraud are minor concerns. In national political elections, however, privacy, reliability and security are important issues to consider.

Dr. Michael Ian Shamos issued six commandments that describe the fundamental requirements for electronic voting [22]. They are listed below in decreasing order of importance:

- I. Thou shalt keep each voter's choices an inviolable secret.
- II. Thou shalt allow each eligible voter to vote only once, and only for those offices for which she is authorized to cast a vote.
- III. Thou shalt not permit tampering with thy voting system, nor the exchange of gold for votes.
- IV. Thou shalt report all votes accurately.
- V. Thy voting system shall remain operable throughout each election.

- VI. Thou shalt keep an audit trail to detect sins against Commandments II-IV, but thy audit trail shall not violate Commandment I.

From the requirements stated above, one can propose a set of security properties that may be used to characterize an electronic voting system. The most common properties of electronic voting systems are described below, along with methods that may be used to satisfy these properties [10].

**Privacy** means that no one can see what a voter, or a proper subset of voters, voted for.

Keeping a voter's choice private is important for security against election fraud. If the contents of a vote is visible for people inside the electoral apparatus, votes can be changed or deleted. This can facilitate election fraud from the inside. By securing votes cryptographically, and requiring multiple servers to decrypt votes, this gives a higher security against insider attacks.

Privacy will not be satisfied if one entity in the voting system is able to view the voters' choices, thus some way of separating the votes cast from the voters' identities is needed. Mix-nets, blind signatures, verifiable secret sharing and homomorphic cryptosystems are all different methods of securing privacy in voting systems. A threshold variant of a homomorphic cryptosystem is used as a basis for the cryptographic voting system chosen for this thesis and will be explained in Chapter 3.

**No Double Voting** means that anyone who tries to cheat by voting more than once will be caught with all but a negligible probability

To ensure that voters are unable to vote twice, some sort of control mechanism must be implemented. This may be satisfied in a variety of ways. The easiest, and perhaps most logical, solution is to maintain a list of people who have voted, and check this list whenever a voter casts his vote. The list could consist of usernames or public keys depending on the authentication mechanism used.

**No Cheating** means that that anyone casting an illegal vote, will be caught with all but a negligible probability.

A way of determining whether a vote is correctly formed is needed. This may be done by having votes digitally signed by an honest on-line signer, or by proving the correctness of votes using zero-knowledge proofs. The latter

is used in the voting system based on homomorphic encryption and will be discussed further in Section 3.3.

**Correctness** means that the result of the election is consistent with the votes cast.

To ensure correctness of the election, one needs to prove that every vote is correctly formed, and that the actions of the servers calculating the results are executed correctly. The voting system should provide some sort of audit trail where the actions of every participant in the election may be verified.

**Verifiable** means that anyone can verify that the result is correct and that every voter may verify that his vote is included in the result.

This property enables everyone to verify the actions of every participant in the election, as well as the actions of the voting system itself. In the voting system based on the homomorphic cryptosystem, a public bulletin board is used where all encrypted votes and proofs of correctness for both voters and servers are posted. These proofs are verifiable for everyone, without requiring the knowledge of any secret key.

**Off-line** means that no specific server or group of servers need to be on-line during the whole election.

An off-line voting system does not need any specific servers to be on-line during the whole election. Whether this is avoidable or not depends largely on the choice of voting system, as well as the particular implementation of the system. Most systems need some sort of on-line server to register votes. Some types of voting schemes may also need additional on-line components, for instance the blind signature scheme which needs an on-line signer to digitally sign votes.

**receipt-freeness** means that a voter cannot prove to a potential coercer that he voted in a particular way.

receipt-freeness can be obtained by securing that no information held by the voter can prove the voter's choice in the election. If we assume that the voter wishes to cooperate with a coercer, receipt-freeness guarantees that such cooperation will not be worthwhile, because it will be impossible for the coercer to obtain proof about how the voter voted. receipt-freeness is a similar property to privacy, with the additional assumption that the voter cooperates with a coercer. Thus, receipt-freeness implies privacy.

receipt-freeness can discourage vote-buying and coercing in an election. When encrypting votes using a randomized cryptosystem, random numbers are used to hide the plaintext. A user-chosen randomness can work as a receipt, as is the case when using a homomorphic cryptosystem for designing an electronic voting system.

**Robustness** means that the voting system can withstand attacks, and can tolerate failure of components.

A voting system should be operational during the whole election and withstand a wide range of attacks. Protection against Denial of Service attacks can be done by replicating critical components in the system. Good authentication mechanisms should be implemented as a protection against hackers.

The properties satisfied by the voting system constructed in this thesis will be discussed in Section 4.6.

## 2.3 Requirements for elections at NTNU

### 2.3.1 Functional Requirements

The elections at NTNU are carried out using Internet voting. The voting service should be accessible from any computer connected to the Internet (and should support all major operating systems).

Many elections at NTNU are carried out using ranked voting, which allows voters to cast a predefined number of differently weighted votes. The system should support this type of election.

### 2.3.2 Security Requirements

The security requirements for elections at NTNU are not as strict as the security requirements for national, political elections. Today's electronic voting system at NTNU satisfies few of the security properties of a secure electronic voting system.

The *privacy* of votes in NTNU elections should be maintained. However, the implications of revealing the contents of votes 5 to 10 years forward in time are assumed to be of little significance (as opposed to national, political elections where votes

ideally should be kept secret for all time). The voting schemes treated in this thesis use cryptography for securing the privacy of votes, and the security of cryptography depend largely on the length of the keys used for encryption. As an increasing key length increases the time needed for encryption, validation and decryption of votes, choosing a suitable key length is important for the performance of the system. The key sizes should be chosen long enough to keep the votes secure today, but the implications of breaking the cryptography 10 years forward in time are not as severe as in the national, political elections. It is assumed that securing the *privacy* of votes 5 years forward in time is sufficient for election at NTNU.

*receipt-freeness* should not be a necessary requirement for this electronic voting system, as vote-buying and coercion are minor concerns at NTNU. In order to understand why *privacy* and *receipt-freeness* are not equally important a few scenarios may be described. If the administrators of the electronic voting system (for instance the administrators of the computer systems at NTNU) may benefit from the election of a specific candidate, it may be desirable for them to tweak the results of the election to their advantage. People with access to the system can easily delete or change preferred votes without being detected, if the contents of the votes are visible. These employees may also be paid by another group or individual at NTNU to change the result of the election. If *privacy* is not satisfied by the system, cheating from the inside may be easy. *Receipt-freeness* is a property discouraging vote-buying and coercion in an election. It is a much harder task to cheat in an election through the purchase of individual votes. This requires the people wanting to cheat in an election to purchase the votes of many voters, and check their receipts to verify that they voted for the correct candidate. Such large scale cheating operations are assumed not likely to be seen at elections at NTNU, and although *receipt-freeness* is needed in large scale political elections, it is a property that is of minor importance for elections at NTNU.

Although it is not provided for elections today, the elections at NTNU could be made universally *verifiable*. This could convince all voters that the results are valid (although electoral fraud has never been an issue with elections at NTNU). A *verifiable* election scheme can ensure people's trust in the political system at NTNU.

A completely *off-line* voting system should not be necessary for elections at NTNU, because the operation of on-line components can be secured using server replication techniques and well-designed authentication mechanisms. However, the com-

plexity and vulnerability of the system will increase with an increasing number of on-line components. It is therefore desirable to keep the number of on-line components to a minimum. The operation of all on-line components needs to be secured for satisfying the *robustness* criterion.

The requirements of *no cheating*, *no double voting* and *correctness* must be satisfied in an electronic voting system at NTNU, because it is desirable to prevent voters from cheating, and to ensure that the results of the election are consistent with the votes cast.

The security and functional requirements for elections at NTNU are summarized in Table 2.1.

Privacy	X
No Double Voting	X
No Cheating	X
Correctness	X
Verifiable	(X)
Off-line	
Receipt-freeness	
Robustness	X
Ranked Voting	X
Internet Voting	X
Usable in a heterogeneous environment	X

Table 2.1: Requirements for a voting system for elections at NTNU.

## 2.4 Existing Electronic Voting Schemes

Various fundamentally different approaches to electronic voting are known in the literature. This section will describe the basics of four different cryptographic voting schemes. Depending on the implementation, all the systems mentioned in this section can be made to satisfy the requirements *robustness* and *no double voting*. These are requirements largely dependent of the implementation of the voting scheme, and the authentication and access control mechanisms used by the voting system.

### 2.4.1 Homomorphic Cryptosystems

A homomorphic cryptosystem is a public key cryptosystem, that satisfies

$$D(c_1) + D(c_2) = D(c_1 \odot c_2)$$

for some binary operation  $\odot$  on ciphertexts. Homomorphic cryptosystems can be used because of this property to calculate the results of an election before decrypting the results, thereby hiding the individual choices of voters. The  $\odot$  function is normally multiplication as in Damgård, Jurik and Nielsen's generalized variant of Paillier's public-key system, which is the cryptosystem used in this thesis. The protocol works in the following way:

1. The voter selects a vote  $v_i$ , chooses a random number  $r$  and calculates the encryption  $E(v_i, r)$ . The voter then makes a proof of correctness that will show that the vote is valid.
2. All votes are multiplied together.
3. The decryption servers make a verifiable threshold decryption, as well as making a proof of correctness of the calculation. When enough decrypted shares are acquired, the shares can be combined to reveal the results of the election without any knowledge of the secret keys.

This voting scheme satisfies the requirements for *privacy*, *no cheating* and *verifiability*. This method is the one studied in detail in this thesis and the functionality and design of this voting system is thoroughly described in Chapter 3. Such voting schemes can also be designed to satisfy *receipt-freeness*, as proposed by Aquisti in [1].

### 2.4.2 Mix-Nets

This voting scheme is based on random order permutations of votes to hide the individual choices of voters. A group of servers permute a set of votes in a random order. If one of the servers is not watched by an adversary, the votes will be randomly permuted and the adversary will be unable to deduce which voter voted what. This is done by making a network of binary gates that either pass the

votes through, or switches them. A server proves that their actions were correct by proving that it did in fact do one of these things at each gate.

This scheme requires some randomized public key cryptosystem and the servers are required to have different keys, to ensure that the permutations of votes remain secret to all servers. This could be done by using the El Gamal cryptosystem and making different exponents for each server. The protocol works in the following way:

1. The voter generates a vote that is encrypted  $k$  times:  

$$c_i = E_{K_k}(R_k, E_{K_{k-1}}(\cdots E_{K_1}(R_1, v_i) \cdots))$$
2. The vote is posted on a bulletin board.
3. The first server decrypts all votes once, makes a random permutation of them and posts the result on the board. Then the rest of the servers do this sequentially.
4. The result of the election is ready after the votes have been decrypted by the last server.

This solution requires many communications rounds and has a large computational overhead, because of the many encryption and decryption operations needed. The *privacy* of votes is obtained by using random permutations performed by the servers. Breaking the scheme would require an adversary to watch the actions of all servers. For obtaining a *verifiable* scheme, large server proofs are needed for proving the *correctness* of the random permutations. By proving that all the servers' actions are valid, *correctness* is also satisfied. Some techniques for obtaining *receipt-freeness* in mix-net based voting systems can be found in [2].

### 2.4.3 Blind Signatures

The idea of this scheme is that an on-line trusted signer digitally signs the votes in a fashion so that the signer cannot see what he has signed. The protocol is as follows:

1. The voter generates a vote  $v_i$ , modifies this to get  $b_i$ . This must be done in such a way that the voter is able to generate a signature on  $v_i$  when a signature on  $b_i$  is received.

2. The voter gets  $b_i$  signed by an trusted on-line signer.
3. The voter checks the signature and generates the signature on  $v_i$  from the signature on  $b_i$ .
4. The voter sends the vote  $v_i$  and the signature to a counter anonymously.
5. The counter checks all signatures and counts the valid votes to get the result.

This scheme requires an on-line signer, and a lot of communication rounds, as well as a way of sending messages totally anonymously. The on-line signer must be on-line during the whole election for providing the signing service, and the counter must be on-line the whole election for receiving the votes. Every voter must communicate with the signer to get the vote signed and then send the vote to the counter. The scheme obtains *privacy* by modifying the vote before sending it to the signer, and also by sending the vote and the signature anonymously to the counter. The *correctness* property is satisfied if the on-line signer is honest, and only signs the valid votes. A voting scheme based on blind signatures that satisfies *receipt-freeness* was proposed in [16].

#### 2.4.4 Publicly Verifiable Secret Sharing

The idea of this voting scheme is to use a Publicly Verifiable Secret Sharing system to generate a secret sharing of the vote to the decryption servers. An explanation of the details of this scheme can be found in [20]. The general protocol is as follows:

1. The voter generates a secret sharing of the vote with one share for each of the servers. The voter also makes a proof of correctness for the vote and sends the value to the decryption servers.
2. The decryption servers checks the proofs and add the shares together, publish the results.
3. The result is the secret of the shares.

This scheme requires secure communication between the voter and the decryption servers and it requires a large block size for the votes, because one share is generated for each of the servers. This scheme satisfies *privacy*, given that all communication is secured. *Correctness* should be satisfied by the proofs provided by the

servers. The scheme is also *verifiable* for anyone if the proofs of correctness are published.

## 2.5 Summary

This chapter described the basics of electronic voting systems and some of the problems of current systems. A set of criteria that can be used for characterizing electronic voting systems was defined. It was also explained how these criteria can be satisfied in an electronic voting system. The requirements regarding electronic voting here at NTNU were explained. Four types of voting schemes were briefly explained. The next chapter will thoroughly explain the voting system based on homomorphic encryption.



## Chapter 3

# The Cryptographic Voting System

This chapter will present the theory behind a voting system based on homomorphic encryption, and explain all building blocks needed to build a fully functional voting system that satisfies the requirements for NTNU elections.

### 3.1 Introduction

The cryptosystem of choice is the Damgård, Jurik and Nielsen's generalization of Paillier's public-key system [19]. This cryptosystem has properties that are attractive for electronic voting. The cryptosystem is homomorphic, which gives the possibility of adding encrypted votes without decrypting them. Another attractive property is the possibility of making a threshold variant of this cryptosystem to ensure that a certain percentage of the decryption authorities must cooperate to decrypt the results of an election. A set of zero-knowledge proofs will be used for proving the correctness of votes. The cryptosystem and the zero-knowledge proofs are the building blocks we need to construct a fully functional and efficient voting system.

### 3.2 Damgård, Jurik and Nielsen's generalized variant of Paillier's public-key system

The Paillier cryptosystem is a probabilistic asymmetric algorithm for public key cryptography, invented by Pascal Paillier in 1999. The scheme is based on composite residuosity classes of degree set to a hard-to-factor number  $n = pq$  where  $p$  and  $q$  are two large prime numbers.

The scheme is an additive homomorphic cryptosystem which implies that:

$$D(c_1 * c_2) = D(c_1) + D(c_2)$$

where  $c_1$  and  $c_2$  are encryptions of the plaintexts  $m_1$  and  $m_2$ . By using this we can, given only the public-key and the ciphertexts  $c_1$  and  $c_2$ , compute an encryption of  $m_1 + m_2$  by multiplying the ciphertexts. This cryptosystem uses computations  $n^2$  where  $n$  is a RSA modulus.

The generalization of Paillier's public-key system proposed by Damgård, Jurik and Nielsen in [19] uses computations  $n^{s+1}$ , where  $n$  is an RSA modulus and  $s$  is a natural number. This cryptosystem contains Paillier's scheme as a special case by setting  $s = 1$ . For every natural number  $s$  and RSA modulus  $n$  we can build a cryptosystem  $CS_s$  with message space  $\mathbb{Z}_n^s$ . This generalization allows the extension of the message space by increasing  $s$  without increasing the keylength, thus without affecting the security of the system.

Both these cryptosystems rely on the difficulty of solving the problem known as the *Composite Residuosity Class Problem*. This is the problem of distinguishing the set of  $n$ -residues from non  $n$ -residues in  $\mathbb{Z}_{n^{s+1}}$

**Definition.** A number  $z$  is said to be an  $n$ 'th residue modulo  $n^{s+1}$  if there exists a number  $y \in \mathbb{Z}_{n^{s+1}}^*$  such that

$$z = y^n \text{ mod } n^{s+1}$$

The assumption that this problem is polynomial-time intractable is referred to as the Decisional Composite Residuosity Assumption (DCRA).

A thorough description of the Paillier public-key cryptosystem can be found in [18], and the generalization of this cryptosystem is explained in [19]

The next subsections will describe the generalized Paillier cryptosystem that will be used in this voting system.

### 3.2.1 Key Generation

In the following,  $gcd$  denotes the greatest common divisor of two numbers, and  $lcm$  denotes the least common multiplier of two numbers. The security parameter  $k$  denotes the bitlength of the admissible RSA modulus  $n = pq$ . The parameters  $p$  and  $q$  are chosen as two odd primes such that  $gcd(n, \phi(n)) = 1$  where  $\phi$  is Euler's totient-function. This function is defined as the number of positive integers less than  $n$  that are relatively prime to  $n$ . The parameter  $g$  is chosen as an element of the group  $\mathbb{Z}_{n^{s+1}}^*$ , with an order divisible by  $n^s$ . To simplify matters it is possible to use  $g = 1 + n$  always, without decreasing security. It will be shown in Section 5.5.2 that the choice of  $g = 1 + n$  can facilitate the optimization of performance as well.

Let  $\lambda$  be  $lcm(p - 1, q - 1)$ . It is now possible to choose a  $d$  such that  $d \bmod n \in \mathbb{Z}_n^*$  and  $d = 0 \bmod \lambda$ . Any  $d$  satisfying these requirements will work, and the original Paillier scheme used  $d = \lambda$ , but there are better choices for the threshold scheme this voting system will be based on. This will be discussed in Section 3.6.

The public key is  $(n, g)$ , the private key is  $d$ , and the message space for plaintexts in the cryptosystem is  $\mathbb{Z}_{n^s}$ .

### 3.2.2 Encryption

Given a plaintext message  $m$ , choose a random  $r \in \mathbb{Z}_n^*$  and calculate the ciphertext as  $E(m, r) = g^m r^{n^s} \bmod n^{s+1}$ .

### 3.2.3 Decryption

Given a ciphertext  $c$ , compute  $c^d \bmod n^{s+1}$ . Decryption is here shown below using  $g = 1 + n$ .

$$c^d = (g^m r^{n^s})^d = ((1 + n)^m r^{n^s})^d = (1 + n)^{md \bmod n^s} (r^{n^s})^d \bmod \lambda = (1 + n)^{md \bmod n^s}$$

A method for extracting a number  $i \bmod n^s$  from  $(1 + n)^{i \bmod n^s}$  was proposed in [19], and will be explained here. We define a function  $L$  as  $L(b) = (b - 1)/n$ . By

applying this function to the value  $(1 + n)^{i \bmod n^s}$  we obtain

$$L((1 + n)^i \bmod n^{s+1}) = (i + \binom{i}{2}n + \cdots + \binom{i}{s}n^{s-1}) \bmod n^s$$

The value  $i$  may now be extracted part by part. We extract  $i_1 = i \bmod n$ , then  $i_2 = i \bmod n^2$  and so forth. It is easy to extract  $i_1 = L((1 + n)^i \bmod n^2) = i \bmod n$ . The rest of the values are extracted by using the following induction step:  $i_j = i_{j-1} + k * n^{j-1}$  for some  $0 \leq k < n$ . Each term  $\binom{i_j}{t+1}n^t$  for  $0 < t < j$  satisfies that  $\binom{i_j}{t+1}n^t = \binom{i_{j-1}}{t+1}n^t \bmod n^j$ . This is due to the fact that the contributions from  $kn^{j-1}$  are cancelled out modulo  $n^j$ , after multiplication by  $n$ . This allows us to calculate  $i_j$  as

$$\begin{aligned} i_j &= i_{j-1} + kn^{j-1} \\ &= L((1 + n)^i \bmod n^{j+1}) - \left( \binom{i_{j-1}}{2}n + \cdots + \binom{i_{j-1}}{j}n^{j-1} \right) \bmod n^j \end{aligned}$$

By using this equation for calculating  $i_j$  the calculation of  $i$  may be done using the algorithm described in pseudocode in listing 3.1.

```

i = 0;
2  for j=1 to s do
    begin
4      t1 = L(a mod n^{j+1});
      t2 = i;
6      for k = 2 to j do
          begin
8          i = i - 1;
            t2 = t2 * i mod n^j;
10         t1 = t1 - \frac{t2^{k-1}}{k!} mod n^j;
            end
12         i = t1;
    end
end

```

Listing 3.1: Algorithm used for extracting  $i \bmod n^{s+1}$  from  $(1 + n)^i \bmod n^{s+1}$

By using the algorithm explained in Listing 3.1 with  $(1 + n)^{md \bmod n^s}$  as input,  $md \bmod n^s$  can be extracted. The plaintext is calculated by multiplying  $md \bmod n^s$  with the inverse of  $d$  modulo  $n^s$ .

### 3.3 Zero Knowledge Proofs

A zero-knowledge proof has the property of proving a statement without yielding anything beyond the validity of the proof. The ability to prove the validity of the votes in a voting system, without revealing the contents is essential to ensure the privacy and integrity of the votes.

The concept of zero-knowledge implies that no matter how the verifier behaves as a verifier, no information can be learned from the conversation with the prover that it could not have computed itself even before the start of the protocol. Protocols satisfying this criteria often requires more than three moves, and in the interest of efficiency a looser criteria has been defined. This is the concept of honest-verifier zero-knowledge. Such a protocol is defined as being zero-knowledge given that the verifier is honest. These protocols often requires fewer moves and are therefore more efficient than the ones satisfying the criteria of zero-knowledge.

To perform these proofs  $\Sigma$ -protocols may be used. These protocols are a type of 3-move honest-verifier zero-knowledge proofs. Consider two parties, a prover  $P$  and a verifier  $V$ . They both share knowledge of a common input  $x$ , and  $P$  knows a witness  $w$  such that  $(x, w) \in R$ , where  $R$  is some relation.  $P$  wants to prove to  $V$  that  $x \in L$  where  $L$  is the NP-language specified by the relation  $R$ , but does not want  $V$  to obtain any other information about  $x$  other than that  $x \in L$ . To prove this,  $P$  and  $V$  may carry out a  $\Sigma$ -protocol.  $P$  sends an initial message  $a$ ,  $V$  responds with a randomly chosen challenge  $e$ , and  $P$  responds with an answer  $z$ . From evaluating  $(x, a, e, z)$ , the verifier can decide to accept or reject the claim that  $x \in L$ .

A proof system like this is called a  $\Sigma$ -protocol when it satisfies the following criteria [6], [9]:

**Completeness:** If  $x \in L$ , and  $P$  knows a witness (the private input to  $P$ )  $w$ ,  $V$  will always accept the proof at the end of the protocol

**Special Soundness:** Given two arguments  $(x, a, e, z)$  and  $(x, a, e', z')$  where  $e \neq e'$ , it is possible to extract a witness  $w$  so that  $(x, w) \in R$ . Special soundness makes a  $\Sigma$ -protocol a system for proofs of knowledge.

**Special honest-verifier zero-knowledge:** Given  $x \in L$  and  $e$ , is it possible to simulate an argument  $(x, a, e, z)$  which is indistinguishable from a real argument with challenge  $e$ . In other words, it is possible to simulate a proof  $(x, a, e,$

$z$ ) with the same probability distribution as real proofs with any witness and conditioned on using the challenge  $e$

An  $\Sigma$ -protocol can be made non-interactive through the Fiat-Shamir heuristic. The challenge will be computed by P as  $e = \text{hash}(x, a)$  instead of being chosen randomly by V. This allows us to complete the whole honest-verifier zero-knowledge proof by sending one message from P to V, thus making it non-interactive. Non-interactive zero-knowledge proofs will be further discussed in Section 3.5.

### 3.4 The Zero-Knowledge Protocols

The different zero-knowledge protocols needed by this system to ensure that votes are correctly formed are explained in this section. The interactive versions of the protocols are described and proved to fulfill the requirements stated in Section 3.3. The protocols explained in the following subsections are not zero-knowledge, only honest-verifier zero-knowledge. This will also be true for the non-interactive variants explained in Section 3.5, as zero-knowledge cannot be obtained using the Fiat-Shamir heuristic. We can however obtain security in the so-called random oracle model. The protocols described in this section were originally proposed in [19].

All these protocols will consist of a conversation between P, a prover, and V, a verifier. Normally the protocols will consist of one message sent from P to V, a reply from V containing a  $q$ -bit random challenge, a message from P containing values calculated using the random challenge, and the verification of the proof at V. All protocols will include a security parameter  $q$ , which denotes the bitlength of the random challenge given by V. This security parameter decide the probability that a cheating prover can make the verifier accept the conversation. This probability is given as  $\leq 2^{-q}$ . For these protocols to satisfy special soundness,  $q$  must be chosen such that  $2^q$  is smaller than the smallest prime factor of  $n$ .

#### 3.4.1 Encryption of 0 - protocol

This protocol is used to prove that a given ciphertext is an encryption of 0, which is equivalent to proving that the ciphertext is a  $n^s$  power. Remember the form of a ciphertext;  $c = E(m, r) = (1 + n)^m r^{n^s} \bmod n^{s+1}$ . If  $(1 + n)^m = 1$  because  $m = 0$ ,

the whole expression is reduced to  $c = r^{n^s} \bmod n^{s+1}$ . Showing that a ciphertext  $c = E(m, r)$  encrypts a given plaintext can be done by showing that  $c(1+n)^{-m}$  is an encryption of 0.

**Common Input:**  $n, u$ .

**Private Input for P:**  $v \in \mathbb{Z}_n^*$ , such that  $u = E(0, v)$ .

1. P chooses  $r$  at random in  $\mathbb{Z}_n^*$  and sends  $a = E(0, r)$  it to V
2. V chooses a random  $q$ -bit challenge  $e$  and sends to P.
3. P sends  $z = rv^e \bmod n$  to V.
4. V then checks that  $u, a, z$  are relatively prime to  $n$  and then computes the encryption  $E(0, z)$ , checks if  $E(0, z) = au^e \bmod n^{s+1}$  and accepts if and only if this is true.

To check the completeness of this proof, the values are inserted into the equation checked by V to ensure that a verifier will accept the conversation, if P knows  $v$ . These calculations are possible due to the homomorphic property of the cryptosystem.

$$E(0, z) = E(0, rv^e \bmod n) = E(0, r)E(0, v)^e \bmod n^{s+1} = au^e \bmod n^{s+1}$$

To prove special soundness we must show that it is possible to extract a witness  $v$  from two accepting conversations  $(a, e, z)$  and  $(a, \hat{e}, \hat{z})$  with  $e \neq \hat{e}$ . From the conversations we have:

$$\begin{aligned} E(0, z) &= au^e \bmod n^{s+1} \\ E(0, \hat{z}) &= au^{\hat{e}} \bmod n^{s+1} \end{aligned}$$

Notice that:

$$E(0, z\hat{z}^{-1} \bmod n) = u^{e-\hat{e}} \bmod n^{s+1}$$

Since  $2^q$  is smaller than the smallest prime factor of  $n$ , it is trivial to see that  $e - \hat{e}$  will be prime to  $n$ . This implies that by using Euclid's algorithm two numbers  $\alpha$  and  $\beta$  can be chosen such that  $\alpha n^s + \beta(e - \hat{e}) = 1$ . Let  $\bar{u} = u \bmod n$  and  $v = \bar{u}^\alpha (z\hat{z}^{-1})^\beta \bmod n$ . We see that  $u^{n^s} \bmod n^{s+1} = E(0, u \bmod n) = E(0, \bar{u})$  from the

encryption algorithm in the cryptosystem. By checking the encryption of  $v$  we see that this protocol satisfies special soundness.

$$E(0, v) = E(0, \bar{u})^\alpha E(0, z\hat{z}^{-1})^\beta = u^{an^s} u^{\beta(e-\hat{e})} = u \bmod n^{s+1}$$

It is easily seen that this protocol satisfies the special honest-verifier zero-knowledge criterion. This is done by checking if a simulated proof given  $u$  and  $e$  will have the same probability distribution as a real proof. P chooses a random  $z \in \mathbb{Z}_n^*$  and sets  $a = E(0, z)u^{-e} \bmod n^{s+1}$ , which is a perfect simulation.

### 3.4.2 Protocol 1-out-of-2 $n^s$ 'th power

The protocol described in the last subsection can be extended to prove that a given ciphertext encrypts one of two plaintexts. This is useful for proving that a vote on a candidate has one of two legal values, without revealing which one it is. P and V know the ciphertext  $c$  and the two candidate plaintexts  $m_1$  and  $m_2$ . Using this, P and V may compute the two values

$$u_1 = \frac{c}{g^{m_1}} \bmod n^{s+1}, u_2 = \frac{c}{g^{m_2}} \bmod n^{s+1}$$

P proves that either  $u_1$  or  $u_2$  encrypt 0, and that he has knowledge of one of the corresponding  $n^s$ 'th roots.

**Common Input:**  $n, u_1, u_2$ .

**Private Input for P:**  $v_1$ , such that  $u_1 = E(0, v_1)$ .

1. P chooses  $r_1$  at random in  $\mathbb{Z}_n^*$ . P then invokes the honest-verifier simulator from the previous subsection on input  $n, u_2$  to get a conversation  $(a_2, e_2, z_2)$ , and sends  $a_1 = E(0, r_1), a_2$  to V.
2. V chooses a random  $q$ -bit challenge  $s$  and sends to P.
3. P then computes  $e_1 = s - e_2 \bmod 2^q$  and  $z_1 = r_1 v_1^{e_1} \bmod n$  and sends  $e_1, z_1$  to V.
4. V then checks that  $s = e_1 + e_2 \bmod 2^q$ ,  $E(0, z_1) = a_1 u_1^{e_1} \bmod n^{s+1}$ ,  $E(0, z_2) = a_2 u_2^{e_2} \bmod n^{s+1}$  and that  $u_1, u_2, a_1, a_2, z_1, z_2$  are relatively prime to  $n$ , and accepts if and only if this is the case.

To see that this proof satisfies the completeness criteria we can insert the values into the equations V checks like in the previous protocol.

From what is calculated by P in step 3 of the protocol, it is easy to see that  $s = e_1 + e_2 \bmod 2^t$ . To check the other values we may use the properties of the homomorphic cryptosystem.

$$\begin{aligned} E(0, z_1) &= E(0, r_1 v_1^{e_1} \bmod n) = E(0, r_1) E(v_1)^{e_1} \bmod n^{s+1} = a_1 u_1^{e_1} \bmod n^{s+1} \\ E(0, z_2) &= E(0, r_2 v_2^{e_2} \bmod n) = E(0, r_2) E(v_2)^{e_2} \bmod n^{s+1} = a_2 u_2^{e_2} \bmod n^{s+1} \end{aligned}$$

To check if this protocol satisfies special soundness we extract the witness  $v$  such that either  $u_1 = E(0, v)$  or  $u_2 = E(0, v)$  from two accepting conversations:

$$(a_1, a_2, s, e_1, z_1, e_2, z_2), (a_1, a_2, \hat{s}, \hat{e}_1, \hat{z}_1, \hat{e}_2, \hat{z}_2)$$

The procedure to check this will be equivalent to the work done in the previous subsection. The difference is that the witness  $v_1$  will show knowledge of only one of the corresponding  $n^s$ 'th roots.

The proof for special honest-verifier zero-knowledge criterion will also be equivalent to what is done in the previous subsection. The simulation will have to be done for each of the corresponding  $n^s$ 'th roots.

### 3.4.3 Protocol 1-out-of- $L$ $n^s$ 'th power

The protocol explained in this subsection extends the protocol from the last subsection to prove in honest-verifier zero knowledge that a ciphertext is an encryption of 1 out of  $L$  plaintexts without revealing which one.

**Common Input:**  $n, u_1, \dots, u_L$ .

**Private Input for P:**  $v_1$ , such that  $u_1 = v_1^{n^s} \bmod n^{s+1}$ .

1. P chooses  $r_1$  at random in  $\mathbb{Z}_n^*$ . P then invokes the honest-verifier simulator of the encryption-of-0 protocol with input  $n, u_i$  to get conversations  $a_i, e_i, z_i$  for all  $i$ 's in  $2 \leq i \leq L$ , and sends  $a_1 = E(0, r_1), a_2$  to V.
2. V chooses a random  $q$ -bit challenge  $s$  and sends it to P.

3. P then computes  $e_1 = s - (e_2 + \dots + e_L) \bmod 2^q$  and  $z_1 = r_1 v_1^{e_1} \bmod n$  and sends  $e_1, z_1, \dots, e_L, z_L$  to V.
4. V then checks that  $s = e_1 + \dots + e_L \bmod 2^q$ ,  $E(0, z_i) = a_i u_i^{e_i} \bmod n^{s+1}$  for all  $i$ 's in  $1 \leq i \leq L$  and that  $u_1, \dots, u_L, a_1, \dots, a_L, z_1, \dots, z_L$  are relatively prime to  $n$ , and accepts if and only if this is the case.

The proofs that this protocol satisfies completeness, honest-verifier zero-knowledge and special soundness follows directly from the 1-out-of-2  $n^s$ 'th power protocol..

### 3.4.4 Protocol Multiplication-mod- $n^s$

This protocol proves in honest-verifier zero-knowledge that a ciphertext contains the multiplication of the plaintexts of two other ciphertexts modulo  $n^s$ . This protocol was described in [19]. The three ciphertexts in question is included in the common input for both P and V as  $e_a, e_b, e_c$ .

**Common Input:**  $n, g, e_a, e_b, e_c$

**Private Input for P:**  $a, b, c, r_a, r_b, r_c$  such that  $ab = c \bmod n$  and  $e_a = E(a, r_a)$ ,  $e_b = E(b, r_b)$ ,  $e_c = E(c, r_c)$

1. P chooses a random value  $d \in \mathbb{Z}_{n^s}$ ,  $r_d, r_{db} \in \mathbb{Z}_n^*$  and sends the encryptions  $e_d = E(b, r_d)$ ,  $e_{db} = E(db, r_{db})$  to V.
2. V chooses a random  $q$ -bit challenge  $e$  and sends it to P.
3. P opens the encryption  $e_a^e e_d = E(ea + d, r_a^e r_d \bmod n)$  by sending  $f = ea + d \bmod n^s$  and  $z_1 = r_a^e r_d \bmod n$ . Finally P opens the encryption  $e_b^f (e_{db} e_c)^{-1} = E(0, r_b^f (r_{db} r_c^e)^{-1} \bmod n)$  by sending  $z_2 = r_b^f (r_{db} r_c^e)^{-1} \bmod n$ .
4. V verifies that the openings of encryptions in step 3 were correct, that all values sent by P are relatively prime to  $n$ , and accepts if and only if this is the case.

To check that this protocol satisfies the completeness criterion the values are inserted into the equations checked by V.

$$\begin{aligned}
 E(f, z_1) &= (g^a r_a^{n^s})^e g^d r_d^{n^s} = e_a^e e_d \\
 E(0, z_2) &= (g^b r_b^{n^s})^{ea+d} (g^{db} r_{db} g^{abe} r_c^{n^s})^{-1} = r_b^f (r_{db} r_c^e)^{-1}
 \end{aligned}$$

The protocol will now be proven to satisfy special soundness. Given two accepting conversations we have  $f = ea + d \bmod n^s$ ,  $\hat{f} = \hat{e}a + d \bmod n^s$ . From this we see that  $fb - x - ec = \hat{f}b - x - \hat{e}c \bmod n^s = 0$ . By putting this together we obtain

$$(f - \hat{f})b = (e - \hat{e})c \bmod n^s \text{ or } (e - \hat{e})ab = (e - \hat{e})c \bmod n^s$$

Because  $(e - \hat{e})$  is invertible modulo  $n^s$  because  $2^q$  is smaller than the smallest prime factor of  $n$ , we can conclude that  $c = ab \bmod n^s$  and also compute  $a, b$  and  $c$ .

For honest-verifier simulation we may choose  $f, z_1, z_2, e$  at random and then computing  $e_d, e_{db}$  which is seen to see a perfect simulation because  $f, z_1, z_2, e$  are random and independent in the real conversation as well.

### 3.4.5 Protocol for Equality of Discrete Logarithms

This protocol explained in [19] proves that two discrete logarithms are equal, and will be used in the decryption process by the decryption servers to convince V that the raising of the ciphertext to their own secret exponent is done correctly, without revealing any secret information about the exponent to V.

**Common input:**  $n, s, u, \tilde{u}, v, \tilde{v} \in \mathbb{Z}_{n^{s+1}}^*$

**Private input for P:**  $y$  such that  $y = \log_u(\tilde{u}) = \log_v(\tilde{v})$

1. P chooses a random number  $r$  of length  $(s+1)k + q$  bits where  $k$  is the bitlength of the modulus, and  $q$  is the security parameter. P then calculates  $a = u^r \bmod n^{s+1}$ ,  $b = v^r \bmod n^{s+1}$  and sends these values to V.
2. V chooses a random challenge  $e$  of  $q$  bits and sends to P.
3. P sends  $z = r + ey$  to V.
4. V verifies that  $u^z = a\tilde{u}^e \bmod n^{s+1}$ ,  $v^z = b\tilde{v}^e \bmod n^{s+1}$  and accepts if and only if this is the case.

A check for completeness is performed by inserting the values into the equations checked by V.

$$\begin{aligned} u^z &= u^r u^{ey} \bmod n^{s+1} = a(u^y)^e \bmod n^{s+1} = a\tilde{u}^e \bmod n^{s+1} \\ v^z &= v^r v^{ey} \bmod n^{s+1} = a(v^y)^e \bmod n^{s+1} = a\tilde{v}^e \bmod n^{s+1} \end{aligned}$$

For verifying that the protocol satisfies special soundness, remember that  $2^q$  is smaller than the smallest of the prime factors of  $n$ . This implies that with two accepting conversations  $(a, b, e, z)$ ,  $(a, b, \hat{e}, \hat{z})$  where  $e \neq \hat{e}$  we have that  $\gcd(e, \hat{e}) = 1$ . Notice that we in this case, like in Section 3.4.1 can choose  $\alpha$  and  $\beta$  such that  $\alpha n^s + \beta(e - \hat{e}) = 1$

$$u^z(u^{\hat{z}})^{-1} = \tilde{u}^e \tilde{u}^{-\hat{e}} \bmod n^{s+1} = \tilde{u}^{e-\hat{e}} \bmod n^{s+1} = u^{y(e-\hat{e})} \bmod n^{s+1} = u^y \bmod n^{s+1}$$

For honest-verifier simulation a random  $z \in \mathbb{Z}_n^*$  is chosen. Given  $e, u, \tilde{u}, v, \tilde{v}$  It is possible to set  $a = \tilde{u}^{-e} u^z$  and  $b = \tilde{v}^{-e} v^z$  which will have the same probability distribution as the real proof.

### 3.5 Non-Interactive Zero-Knowledge Proofs

All the protocols described in the previous section are interactive. As interactive protocols require communication between P and V, they require on-line servers to reply with random values to complete the proofs. By allowing P to choose the random value while computing the values for the proofs, we can relieve P of interacting with V, thus reducing these 3-way protocols to non-interactive proofs. This may be done by using the Fiat-Shamir heuristic and a suitable hash function  $H$ . This variant of the proofs is not only more efficient due to less communication between P and V, but also allows V to be off-line and later check the proof created by P. The non-interactive zero-knowledge proofs will only provide security in the random oracle model.

#### 3.5.1 Random Oracle

The random oracle is a mathematical abstraction used in cryptographic proofs. In practice, random oracles are typically used to model cryptographic hash functions in schemes where strong randomness assumptions are needed of the hash function's output, as is the case in the zero knowledge protocols described in the previous section. Note however, that the non-interactive variants of these proofs are not zero-knowledge as they stand, but provide security assuming the random oracle model. In the more precise definition formalized in [3], the random oracle produces a bit-string of infinite length which can be truncated to the length desired. When

a random oracle is used within a security proof, it is made available to all players, including the adversary.

No real function can implement a true random oracle however, and although a protocol is proven secure in the random oracle model, it may turn out to be trivially insecure when the random oracle is replaced with a real hash function. Nonetheless, a proof of security in the random oracle model gives strong evidence that an attack which does not break the other assumptions of the proof, must discover some unknown and undesirable property of the hash function used in the protocol in order to work.

### 3.5.2 Fiat-Shamir Heuristic

The interactive protocols explained in Section 3.4 consist of three messages between  $P$  and  $V$  ( $P \rightarrow V$ ,  $V \rightarrow P$ ,  $P \rightarrow V$ ), where the message from  $V$  to  $P$  consists of a random challenge. Fiat and Shamir present a zero-knowledge identification scheme in [8], where they prove that  $V$ 's part in this protocol may be replaced by a truly random function  $f$ , thereby making the scheme non-interactive. This will imply, as mentioned above, that the scheme no longer is truly zero-knowledge, but will provide security assuming the random oracle model.

The Fiat-Shamir heuristic applies to any 3-round zero-knowledge protocol, and can be used in our case to transform the interactive zero-knowledge protocols into valid proofs assuming the random oracle model. We can however never design a truly random algorithm  $f$ . This requires that the function  $f_0$  that is used in practice must be sufficiently strong to withstand attacks on the scheme.

### 3.5.3 The Non-Interactive Zero-Knowledge Proofs

The protocols are made non-interactive by replacing the random value chosen by  $V$ , with a hash function  $H$ . The digest should be calculated from values that are unique for the specific proof. Therefore the digest should be calculated over the values present in the argument presented to  $V$ . In addition to the values used in the argument, some auxiliary information should be included in the input to  $H$ , to prevent duplication of the proof. This information can for instance include the *id* of the voter, as well as the modulo  $n$  used in the election.

The non-interactive variant of the encryption-of-0 protocol will be described here

as an example. The non-interactive variants of the other protocols described in Section 3.4 are trivial extensions to this example:

**Common Input:**  $n, u, id$ .

**Private Input for P:**  $v \in \mathbb{Z}_n^*$ , such that  $u = E(0, v)$ .

1. P chooses  $r$  at random in  $\mathbb{Z}_n^*$  and computes  $a = E(0, r)$  to V
2. P then calculates random  $q$ -bit challenge  $e$  as  $e = H(n, a, u, id)$ .
3. P sends  $z = rv^e \bmod n, a$  to V.
4. V then calculates  $e = H(n, a, u, id)$ , checks that  $u, a, z$  are relatively prime to  $n$  and then computes the encryption  $E(0, z)$ . V then checks if  $E(0, z) = au^e \bmod n^{s+1}$  and accepts if and only if this is true.

By leaving to V the calculation of  $e$  from the information used in the proof, V can be certain that the  $e$  used in the calculations made by P is correct. If this is not the case, this will be discovered when verifying that  $E(0, z) = au^e \bmod n^{s+1}$ .

### 3.6 A Threshold Variant of the Cryptosystem

A threshold variant of the cryptosystem should be implemented to ensure that, given  $l$  decryption servers, only a subset of at least  $w$  authentication servers can be able to decrypt votes. This reduces the risk for voting fraud committed by persons in the election apparatus and secures the privacy of the votes. Damgård, Jurik and Nielsen proposed a threshold variant of their cryptosystem in [19], where it is proven in the random oracle model that this threshold scheme is as secure as a centralized scheme, where one trusted entity performs the decryption. The threshold scheme uses a variant of Shoup's threshold variant of RSA signatures [23]. Shoup's variant works by allowing a set of servers to collectively raise an input number to a secret exponent modulo  $n$  (a RSA modulus.) Each server returns a share of the result, together with a proof of correctness. When  $w$  shares are acquired, these can be combined to reveal the result.

This protocol can, as explained in [19] be transplanted to the cryptosystem used in this voting system, thus allowing the decryption servers to collectively raise an

input number to the power of the secret exponent  $d \bmod n^{s+1}$ . The protocol can therefore be used to compute  $E(m, r)^d \bmod n^{s+1} = c^d \bmod n^{s+1}$ . After doing this the rest of the decryption process may be done by anyone without knowing the secret key  $d$ . The security of the scheme will depend on the choice of  $d$ , because if the original choice  $d = \lambda$  is used, then seeing the value  $E(m, r)^d \bmod n^{s+1}$  may allow an attacker to compute  $\lambda$ , thus breaking the system completely. Therefore  $d$  should be chosen as a number different from  $\lambda$ , such that  $d = 1 \bmod n^s$  and  $d = 0 \bmod \lambda$ . This will ensure that  $d$  contains no trace of the secret  $\lambda$ .

The parameter  $w$  can be chosen freely, but a natural choice would be  $w = \frac{l}{2}$ . The modulus  $n$  will be a product of safe primes  $p, q$ , where  $p, q, p' = \frac{(p-1)}{2}, q' = \frac{(q-1)}{2}$  are primes. As mentioned earlier the decryption servers have to include a proof that their exponentiation of the ciphertext is done correctly. The protocol designed for this purpose is proposed in [19], and is described in Section 3.4.5.

### 3.6.1 Key Generation

As explained earlier, the key generation process starts by choosing two safe primes  $p$  and  $q$ . These primes are safe if they satisfy  $p = 2p' + 1$  and  $q = 2q' + 1$  where  $p'$  and  $q'$  are prime. The straight forward way of finding safe primes is choosing  $p$  and  $q$  at random at first, and then testing whether or not they have the needed properties. A more efficient way of generating safe primes is explained in [13]. After choosing  $p$  and  $q$  we set  $n = pq$  and  $m = p'q'$ . The parameter  $s$  is chosen to get the desired message space for the election. The parameters  $n$  and  $s$  give the plaintext space  $\mathbb{Z}_n^s$ . By increasing  $s$  the message space can be increased without increasing the key length  $k$ , and therefore without affecting the security of the scheme. The secret key  $d$  is chosen such that  $d = 0 \bmod m$  and  $d = 1 \bmod n^s$ .

Now we go on to generating the secret key shares for the decryption servers. The secret key share for decryption server  $s_i$  will be chosen as  $s_i = f(i) = \sum_{j=0}^{w-1} a_j i^j \bmod n^s m$  where  $a_j$  for all  $1 \leq j < w$ , are chosen as random values from  $0, \dots, n^s(m-1)$ , and  $a_0 = d$ . This gives a secret key share for decryption server  $i$ ,  $s_i$ , and a public key  $(n, g)$ . To be able to verify the actions of each decryption server the following fixed public values are needed:  $v$  generating the cyclic group of squares in  $\mathbb{Z}_{n^{s+1}}^*$  and a verification key for each decryption server  $v_i = v^{\Delta s_i} \bmod n^{s+1}$  where  $\Delta = l!$

### 3.6.2 Encryption

For encrypting a message  $M$  a random  $r \in \mathbb{Z}_n^*$  is picked and the ciphertext is computed as  $c = (n + 1)^M r^{n^s} \bmod n^{s+1}$  as in the normal scheme described in Section 3.2.2. Some improvements regarding the efficiency of this computation will be discussed in Section 5.5.2.

### 3.6.3 Share Decryption

The  $i$ 'th decryption server performs its share decryption by computing  $c_i = c^{2\Delta s_i}$ , where  $c$  is the ciphertext and  $s_i$  is the secret key share. Together with  $c_i$ , the decryption server will include a proof using the protocol explained in Section 3.4.5, showing that  $\log_{c^4}(c_i^2) = \log_v(v_i)$ , which will convince a verifier  $V$  that the ciphertext was raised to the power of the secret exponent  $s_i$ .

### 3.6.4 Share Combination

When the  $w$  shares from the decryption servers are acquired and all proofs have been checked, the results can be combined to yield the plaintext without any knowledge of any secret key. The results are obtained by first calculating

$$c' = \prod_{i \in S} c_i^{2\lambda_{0,i}^S} \bmod n^{s+1} \text{ where } \lambda_{0,i}^S = \Delta \prod_{i' \in S \setminus i} \frac{-i}{i - i'} \in \mathbb{Z}$$

The value of  $c'$  will have the form  $c' = c^{4\Delta^2 f(0)} = c^{4\Delta^2 d}$ . We note that  $4\Delta^2 d = 0 \bmod \lambda$  and  $4\Delta^2 d = 4\Delta^2 \bmod n^s$  so  $c' = (1 + n)^{4\Delta^2 M} \bmod n^{s+1}$  where  $M$  is the plaintext. By using the algorithm described in Section 3.2.3 we may extract  $4\Delta^2 M$ , and multiply with the inverse of  $4\Delta^2 \bmod n^{s+1}$  to obtain the plaintext message  $M$ .

## 3.7 Summary

This chapter explained the building blocks needed for designing a fully functional and efficient voting system. The underlying homomorphic cryptosystem has been thoroughly explained, along with the zero-knowledge protocols needed to prove correctness of votes. These protocols has been made non-interactive with help of the Fiat-Shamir heuristic, with security in the random oracle model. A threshold

variant of the cryptosystem has been explained, which is the cryptosystem used for designing a fully functional voting system in the following chapter.



## Chapter 4

# Design

This chapter will describe the design of the voting system and find suitable values for the system parameters to satisfy the computational requirements, as well as the security requirements, for elections at NTNU. Chapter 3 explained the building blocks needed to design the voting system. It is now time to combine these pieces to create a fully functional voting system.

### 4.1 Top Level Architecture

The voting system consists of four main parts; the election authority, the decryption servers, the bulletin board and the voter application. The election authority is responsible for creating the keys needed for encryption and decryption of votes, as well as the calculation of the result from the multiple shares acquired from the decryption servers. The decryption servers decrypt the votes together, using multi-party computations. The bulletin board is the public place where votes are posted. The voter application is responsible for the encryption of the voters' choices and the creation of the appropriate proofs. In order to counter denial of service attacks, multiple bulletin boards may be created, but as this is a trivial extension to the system, only one bulletin board will be used in the design described in this thesis. The top level architecture of the system is shown in Figure 4.1.

The application will typically consist of multiple layers. A possible structure of an implementation of the system in Java is shown in Figure 4.2.

This thesis focuses mainly on the design of the voting system. Although securing

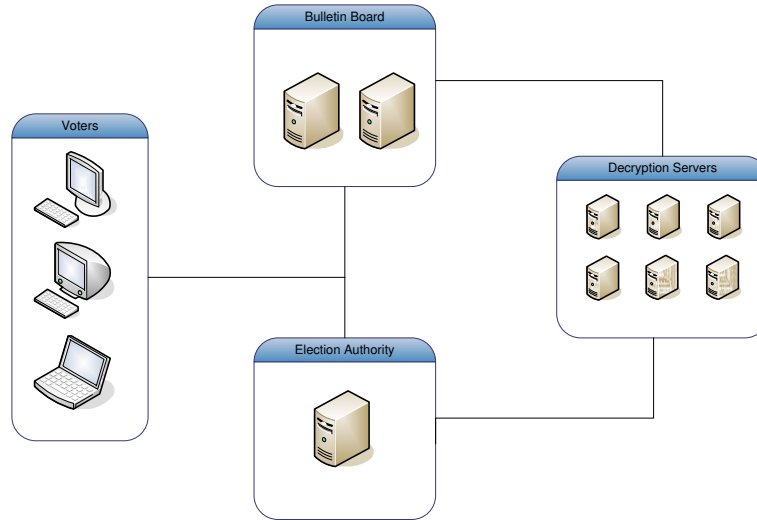


Figure 4.1: Top level architecture of the voting system.

communication between components is outside the scope of this thesis, the design of this system will try to suit the needs of NTNU elections. The internal network at NTNU<sup>1</sup>) may be used for authentication of voters. The communication with this internal network is secured using SSL<sup>2</sup>. The username of the voter can be used for identification and to prevent double voting.

The system will be verifiable for anyone participating in the election. All the zero-knowledge proofs are publicly accessible. This way anyone can verify the proofs of correctness posted on the bulletin board by both voters and decryption servers.

The GUI layer of the model will show the voter the ballot and take input from the voter. The encryption of the votes, and the creation of the proofs of correctness will be done in the voting system layer of the model, using the techniques described in Section 3.2. In this particular model, the networking part of the system is designed as a distributed system and is handled by Java RMI<sup>3</sup> over an SSL connection using

<sup>1</sup>The internal network used by employees and students at NTNU is called Innsida. Innsida carries out authentication of users using a username and password [15].

<sup>2</sup>Secure Sockets Layer. More information can be found at [17]

<sup>3</sup>Java Remote Method Invocation. For more information on Java RMI see [11]

JSSE<sup>4</sup>.

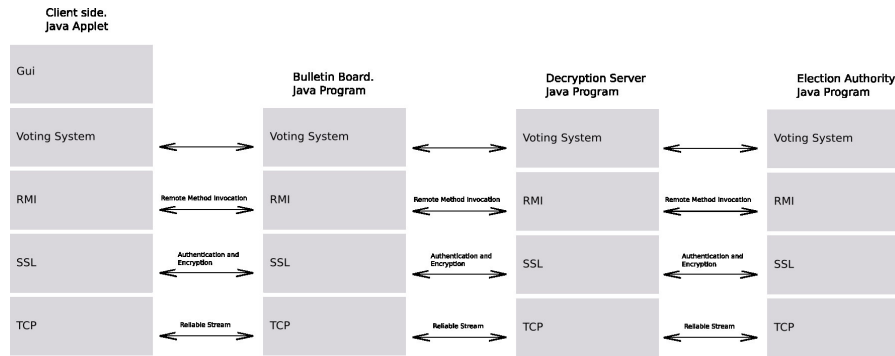


Figure 4.2: Possible layered structure of the system

## 4.2 Parameters of the Election

Table 4.1 presents the parameters used in the calculations of the electronic voting system. This notation will be used throughout the thesis.

$n$	The modulo used in the cryptosystem
$g$	The generator used in the cryptosystem
$k$	The bitlength of $n$
$s$	The exponent used in the cryptosystem to adjust the size of the message space
$q$	The security parameter specifying the length of the challenges in the zero-knowledge proofs
$L$	The number of candidates in the election
$M$	The maximum number of voters in the election
$l$	The number of decryption servers
$w$	The number of decrypted shares needed to get the plaintext of the results from the election

Table 4.1: The parameters of the voting system.

## 4.3 Model of Elections

The general model of elections used in [19] will be explained here and used throughout the thesis.

<sup>4</sup>Java Secure Socket Extension (JSSE) For more information on JSSE see [12]

The model consists of a set of voters  $V_1, \dots, V_M$  where  $M$  denotes the maximum number of voters in the election, the bulletin board  $B$  where votes are posted, and a set of decryption servers  $A_1, \dots, A_l$  where  $l$  is the number of decryption servers. The bulletin board is assumed to be public, and every voter is allowed to write to the board once and only once. A vote is not allowed to be deleted once it is cast, and every voter can see who cast every vote.  $H$  will denote a fixed hash function used to make non-interactive proofs according to the Fiat-Shamir heuristic, with security in the random oracle model as explained in 3.5. An instance of the threshold version of the cryptosystem with public key  $(n, g)$  is set up as explained in 3.6. The  $n$  and  $s$  will be chosen such that  $n^s > M^L$ .

The notation  $Proof_V(S)$ , where  $S$  is some logical statement, will denote a bit string created by voter  $V$  to prove the statement. Proofs are constructed by selecting the appropriate protocol from Section 3.4, and following the steps explained in Section 3.5 to make it non-interactive.

## 4.4 The Different Parts of the Voting System

This section will describe the individual parts of the voting system.

### 4.4.1 The Voter Application

The voter application is the part of the voting system where the voter makes his choice in the election. The vote is encrypted, the zero-knowledge proofs of correctness are created, and the vote is submitted to  $B$ . This part of the voting system can be implemented as an applet or a web service for Internet voting. The voters may be authenticated and authorized to vote through the local intranet (in this case through Innsida) using their normal credentials.

The applet must acquire the election parameters and the public key for the voting system from the election authority. This information can either be included with the application or obtained on-demand from the election authority. The problem of distributing keys will not be discussed further in this thesis, as adequate solutions for solving this problem exist today.

The voter application gets the choices of the user from the user interface and retrieves the information needed to encrypt the votes from the election parameters. It

then follows the correct protocol to create the vote and the proofs needed, as will be explained in Section 4.5. Finally, the application authenticates to  $B$  and submits the votes and the proofs.

#### 4.4.2 The Bulletin Board

The bulletin board is the least complex part of the system. It has to authenticate users, and handle the storage of votes and proofs on the public bulletin board. It may also authenticate servers and post the results of the share decryption, as well as the proofs of correctness related to this computation. The storage of these values may be realized with a database. The values posted on the board should be accessible for every participant of the election. This component is the only one required to be on-line for the whole duration of the election. Denial of service attacks can be a threat to the operation of  $B$ , and a remedy may be the introduction of server replication to secure the operation of this component.

#### 4.4.3 The Decryption Server

The function of the decryption servers is to verify the proofs of the votes cast and provide a share decryption of the result, as well as including a proof that this is performed correctly. The key distribution problem has to be solved for these servers as well because they all need a secret decryption key to perform the share decryption as explained in Section 3.6.3.

The verification of the zero-knowledge proofs will be carried out as explained in Section 3.4. Depending on the type of election, different proofs are used for proving correctness of votes. Section 4.5 will describe the different voting methods thoroughly.

The decryption server will raise the product of the valid votes to the power of the secret key  $s_i$ , and the proof of correctness will together with the public verification key  $v_i$  prove that the computations are done correctly. By proving in honest-verifier zero-knowledge that  $\log_{c^4}(c_i^2) = \log_v(v_i)$ , using the non-interactive variant of the protocol for equality of discrete logarithms, the decryption server will convince a sceptical verifier that the product of the valid votes in fact were raised to the power of the secret key  $s_i$ .

#### 4.4.4 The Election Authority

The election authority is the trusted component of this system. It generates the public key for the system, along with the secret keys for the decryption servers and the public verification keys used to verify the actions of the decryption servers. The generation of keys in this system is done as explained in 3.6.1.

The election authority may also combine the shares acquired from the decryption servers, verify the proofs of their calculations and combine the shares to yield the result. This is a procedure which anyone may do after enough valid shares have been acquired, without knowledge of any secret key, but the election authority may be a natural choice for the publication of the results.

### 4.5 Voting Methods

This section will explain the different voting methods available using this voting system, and how they are realized.

The basic idea is to encode a vote for candidate  $j$  as an encryption of the number  $M^j$ , where  $0 \leq j < L$ . By multiplying all these encryptions we get an encryption of the form  $a = \sum_{j=0}^L a_j M^j \bmod n^s$ , where  $a_j$  is the number of votes cast for candidate  $j$ . This encoding requires us to choose the parameters such that  $L \log_2 M < (k-1)s$ . This is to ensure that the message space is large enough to avoid overflow when votes are multiplied together. This can always be done by increasing  $k$  or  $s$ .

A vote for one of the  $L$  possible candidates will be denoted  $\tilde{v}$ . In the elections where one voter is allowed to cast  $t$  votes for  $L$  candidates, the  $t$  votes will be denoted  $\tilde{v}_1, \dots, \tilde{v}_t$ .

The amount of work that needs to be done by the voter varies from one method to another, and will be described specifically for each method. The decryption servers multiply all votes before share decryption, thus only one share decryption is needed for each decryption server. The amount of zero-knowledge proofs that needs to be verified will also vary and will be mentioned specifically for each method. All zero-knowledge proofs used in the system are made non-interactive using the Fiat-Shamir heuristic for providing security in the random oracle model.

Java implementations of the vote creation process for the different voting methods

are provided in Section A.1. The implementation of the verification of the different zero-knowledge proofs can be seen in Section A.3.

#### 4.5.1 Yes/No Election

The Yes/No election is the simplest of the voting methods available in this voting system. It requires few computations and few proofs to ensure that the votes are correctly formed. The elections can be viewed as an election with two possible candidates. The protocol for how this system carries out a Yes/No election is described in the following:

1. Each voter  $V_i$  decides on a candidate  $j$  from the two possible choices 0, 1. The voter then encodes the vote as  $\tilde{v} = M^j$ , and calculates the encryption of the vote  $e = E(\tilde{v}, r)$ . The voter creates a proof

$$Proof_V(e/g \text{ or } e/g^M \text{ is an encryption of } 0)$$

using the 1-out-of-2  $n^s$ th power protocol, and writes  $e$  and the proof to  $B$ .

2. Each decryption server  $A_x$  checks the proof of all votes, and multiplies all valid votes together to get  $e_{tot}$ . This is the encryption of the result of the election due to the properties of the homomorphic cryptosystem. Finally,  $A_x$  executes his part of the threshold decryption protocol as explained in Section 3.6.3 with  $e_{tot}$  as input, and sends the result of the exponentiation and the proof to  $B$ .
3. When  $w$  servers have executed Step 2 of this protocol and posted the result with a valid proof at  $B$ , anyone may perform share combination, as explained in Section 3.6.4, to reconstruct the plaintext of the encryption, which is the result of the election.

The complexity of the proof is constant in this case, and a decryption server must validate 2 values for each vote.

The voter only computes one encryption, and only one proof is needed in order to prove the correctness of a vote, in addition to the honest-verifier simulator that needs to be computed in the 1-out-of-2  $n^s$ th power protocol.

### 4.5.2 1-out-of- $L$ Election

The 1-out-of- $L$  election is a generalization of the Yes/No election. The computational overhead will be larger in this type of election due to the fact that the 1-out-of- $L$   $n^s$ 'th protocol will be used. As explained in Section 3.4.3, this requires more computations and  $L - 1$  invocations of the honest-verifier simulator for the  $n^s$ 'th power protocol. The generalization of the Yes/No election to allow  $L$  candidates can now be described:

1. Each voter  $V_i$  decides on a candidate  $j$  from the  $L$  possible choices  $0, \dots, L$ . The voter then encodes the vote as  $\tilde{v} = M^j$ , and calculates the encryption of the vote  $e = E(\tilde{v}, r)$ . The voter creates a proof

$$Proof_V(e/g^{M^0} \text{ or } \dots \text{ or } e/g^{M^L} \text{ is an encryption of } 0)$$

using the 1-out-of- $L$   $n^s$ 'th power protocol, and writes  $e$ , as well as the proof, to  $B$ .

2. Each  $A_x$  checks the proof of all votes, and multiplies all valid votes together to get  $e_{tot}$ . Finally,  $A_x$  executes his part of the threshold decryption protocol with  $e_{tot}$  as input, and sends the result of the exponentiation and the proof to  $B$ .
3. When  $w$  servers have executed Step 2 of this protocol and posted the result with a valid proof at  $B$ , anyone can perform the share combination to obtain the result of the election.

The protocol for this type of election will be similar to the Yes/No election, except that the proof will convince a sceptical verifier that the plaintext value of the vote is one of the  $L$  allowed values for the election.

For voting in a 1-out-of- $L$  election, each voter  $V_i$  is required to compute one encryption, and to create one 1-out-of- $L$   $n^s$ 'th power proof in order to prove the correctness of a vote. This proof has a complexity of  $O(L)$ , because the vote must be proven to contain one of the  $L$  valid candidates, thus requiring the decryption servers to verify  $O(L)$  values to validate a vote.

A voter only computes one encryption, along with one 1-out-of- $L$  proof.

### 4.5.3 $t$ -out-of- $L$ Election

The  $t$ -out-of- $L$  election provides the possibility of casting a vote for  $t$  out of  $L$  candidates. The 1-out-of- $L$   $n^s$ 'th power protocol will be used for each of the  $t$  votes to ensure that every vote given is for one of the  $L$  candidates. In addition to this protocol, we need a way to ensure that a voter does not cast multiple votes for the same candidate. This method can easily be used for providing ranked elections with differently weighted votes without any additional requirements for the message space. The protocol for performing an election with these properties is shown below:

1. The voter  $V_i$  chooses  $t$  candidates, encodes the votes  $\tilde{v}_1, \dots, \tilde{v}_t$ . The voter then creates the encryptions of the votes

$$e_1 = E(\tilde{v}_1, r_1), \dots, e_t = E(\tilde{v}_t, r_t)$$

and the 1-out-of- $L$   $n^s$ 'th power proofs

$$Proof_{1_V}, \dots, Proof_{t_V}$$

as explained in the last section. In addition to these proofs, the voter creates the encryptions of all pairwise differences of votes

$$e_{1,2} = E(\tilde{v}_1 - \tilde{v}_2, r_{1,2}), \dots, e_{t,t-1} = E(\tilde{v}_t - \tilde{v}_{t-1}, r_{t,t-1})$$

and the encryptions of the inverses of these differences

$$e'_{1,2} = E((\tilde{v}_1 - \tilde{v}_2)^{-1}, r'_{1,2}), \dots, e'_{t,t-1} = E((\tilde{v}_t - \tilde{v}_{t-1})^{-1}, r'_{t,t-1})$$

This needs to be performed once for every pair of votes. This implies that the encryption of the pairwise difference of votes and the inverse of this difference must be calculated for an asymmetric set of all possible combinations of two different votes. Alternatively, the calculation of the differences of two votes may be done as follows:

$$e_{1,2} = E(\tilde{v}_1, r_1)E(\tilde{v}_2, r_2)^{-1} \dots e_{t,t-1} = E(\tilde{v}_t, r_t)E(\tilde{v}_{t-1}, r_{t-1})^{-1}$$

The voter then creates an honest-verifier zero-knowledge proof using the multiplication-mod- $n^s$  protocol showing that the product of the pairwise dif-

ference and the inverse of this difference is 1 for every pair of votes. By providing this proof,  $V_i$  can convince a sceptical verifier that the pairwise difference between all votes are non-zero, thus no two votes are the same.

After encrypting the votes and creating the required zero-knowledge proofs, the voter  $V_i$  writes the encrypted votes and the proofs to  $B$ .

2. Each  $A_x$  checks the 1-out-of- $L$   $n^s$ 'th proofs provided by  $V_i$  for each of the votes cast. Each  $A_x$  also needs to verify that the pairwise differences of all votes cast by  $V_i$  are non-zero. The encryptions of the difference of two votes can easily be obtained by the verifier, due to the homomorphic property of the cryptosystem. This difference is computed by the verifier as follows:

$$E_{x,y}(x - y, r_x r_y^{-1}) = E(x)E(y)^{-1} = (g^x r_x^{n^s})(g^y r_y^{n^s})^{-1} = g^{x-y} r_x^{n^s} r_y^{-n^s}$$

The multiplication-mod- $n^s$  proofs provided by  $V_i$  are then checked to verify that the multiplication of the encryptions  $e_{x,y}$  and  $e'_{x,y}$ , for all pairwise differences of votes, equals the encryption of 1. If the election is using differently weighted votes, the  $t$  votes provided by  $V_i$  may be exponentiated with the correct weights, before the multiplication of the valid votes. All valid votes acquired from voters are then multiplied together to yield  $e_{res}$ , which is the encrypted result of the election. The encrypted result  $e_{res}$  is raised to the power of the server's secret key  $s_i$ , and a proof of correctness for the exponentiation is made using the proof of equality of discrete logarithms, to prove that  $\log_{c^a}(c_i^2) = \log_v(v_i)$ . The vote and the proof are then sent to  $B$ .

3. Like earlier, the shares provided by the decryption servers may be combined to yield the result of the election.

This protocol will run  $t$  1-out-of- $L$  elections, and prove that no two votes are the same. This solution requires  $O(tL + \frac{t^2}{2} - \frac{t}{2}) = O(tL + t^2)$  zero-knowledge proofs. The number of proofs needed in order to prove the correctness of the votes cast by one voter increases quadratically with an increasing  $t$ .

The voter needs to create  $t$  encryptions of votes, along with the required number of zero-knowledge proofs.

#### 4.5.4 1-out-of- $L$ Election with Binary Encoding of Votes

The 1-out-of- $L$  election with binary encoding of votes provides the same possibilities as the one mentioned in Section 4.5.2, with a possible reduction in the number of proofs needed. The method is explained in [19]. The encoding of a vote for candidate  $j$ , associated with the value  $\tilde{v}$ , will have the form  $M^{\tilde{v}}$ , where  $0 \leq \tilde{v} < L$ . This method uses binary representation of  $\tilde{v}$  to prove the correctness of a vote. An  $l$  is chosen as the smallest possible number such that  $2^{l+1} > L$ . We then let  $b_0, \dots, b_l$  be the bits in the binary representation of  $\tilde{v}$ . We can now see that  $M^{\tilde{v}} = (M^{2^0})^{b_0} \cdot \dots \cdot (M^{2^l})^{b_l}$ . This implies that each factor in this product is either 1 or a power of  $M$ . This may be used in the following protocol, for proving the correctness of a vote:

1. The voter makes the  $l+1$  encryptions  $e_0, \dots, e_l$  of  $(M^{2^0})^{b_0}, \dots, (M^{2^l})^{b_l}$ . For every encryption he also computes

$$Proof_{i_V}(e_i/g \text{ or } e_i/g^{M^{2^i}} \text{ is an encryption of } 0)$$

using the 1-out-of-2  $n^s$ 'th power protocol.

2. Let

$$F_i = (M^{2^0})^{b_0} \cdot \dots \cdot (M^{2^i})^{b_i}$$

for  $i = 1 \dots l$ . The voter then computes the encryptions  $f_i$  of all  $F_i$ , and sets  $f_0 = e_0$ . Now, for  $i = 1 \dots l$  the voter computes

$$Proof_{i_V}(\text{Plaintexts corresponding to } f_{i-1}, e_i, f_i \text{ satisfy } F_{i-1}(M^{2^i})^{b_i} = F_i \text{ mod } n^s)$$

using the multiplication-mod- $n^s$  protocol. The last encryption  $f_l$  is the encryption of the vote.

After encrypting the vote and creating the required zero-knowledge proofs, the voter  $V_i$  writes the encrypted vote and the proofs to  $B$ .

3. The decryption server checks all of the proofs, multiplies all valid votes together, and creates a share decryption of the result.
4. The shares acquired from the decryption servers may now be combined to yield the final result of the election.

These proofs convince a sceptical verifier that  $f_i$  is a number of the form  $M^j$ . Since there are  $l + 1$  encryptions  $e_0, \dots, e_l$ , each determining one bit of  $\tilde{v}$ , it is clear that  $0 \leq \tilde{v} < 2^{l+1}$ . This scheme allows voters to vote for a non-existing candidate  $L < \tilde{v} < 2^{l+1}$  if  $L < 2^{l+1}$ , and it requires a larger block size satisfying  $M^{2^{l+1}} < n^s$ , to avoid overflow when votes are added. Preventing voters from voting for non-existing candidates may be achieved by adding an extra step to the verification of a vote, where the voter proves in zero-knowledge that  $\tilde{v} < L$ . By preventing voters from voting for non-existing candidates this could be reduced to  $M^L < n^s$ . An explanation of how to do this can be found in [19].

This method requires  $O(\log_2 L)$  proofs in order to prove the correctness of a vote, as opposed to  $O(L)$  proofs in the method explained in Section 4.5.2. When using binary encoding of votes, the proofs require more computations than the proofs used in the normal encoding of votes, which may lead to lower performance for small values of  $L$ . For large values of  $L$ , however, the method using binary encoding of votes will perform better than the method explained in Section 4.5.2, due to the lower number of proofs needed.

This method requires the voter to compute  $2l$  encryptions for creating the zero-knowledge proofs, in addition to the  $l$  encryptions required for the encryption of the vote, as opposed to just 1 encryption in the method in Section 4.5.2.

#### 4.5.5 $t$ -out-of- $L$ Election with Binary Encoding of Votes

The  $t$ -out-of- $L$  election with binary encoding of votes is an optimization of the  $t$ -out-of- $L$  election, using binary encoded votes. This is performed by running  $t$  parallel 1-out-of- $L$  elections with binary encoding of votes, and proving that none of the  $t$  votes are cast for the same candidate. The cross validation of votes is performed as explained in Section 4.5.3.

This type of election needs  $O(t \log_2 L + t^2)$  proofs in order to prove the correctness of the votes cast by one voter. This is due to the fact that  $t$  1-out-of- $L$  elections require verification of  $t \log_2 L$  proofs, and the cross validation of all pairwise differences of votes requires verification of  $O(t^2/2 - t/2)$  proofs.

The voter needs to compute  $2lt$  encryptions in order to create the zero-knowledge proofs for the correctness of the votes,  $lt$  encryptions for the encryption of the votes, as well as the encryptions required for cross validation of the votes.

## 4.6 Satisfying Election Requirements at NTNU

### 4.6.1 Functional Requirements

The design of the system facilitates *Internet voting*. By designing the electronic voting system as a distributed system, all components of the system are able to exchange information using remote procedure calls in an Internet environment.

The requirement that the system should be *usable in a heterogeneous environment* will largely depend on the implementation of the system. For the voting service to be accessible using all major operating systems, this application should be implemented using a non-proprietary language, which is independent of the operating system. This system is thought implemented using Java. The majority of operating systems today support Java. This ensures that voters are able to cast votes using most of the operating systems available today.

The proposed system handles both 1-out-of- $L$  elections, where a voter casts only one vote, and  $t$ -out-of- $L$  elections, which may be used to carry out *ranked voting*, where the  $t$  votes may be weighted differently when the result is calculated.

### 4.6.2 Security Requirements

As mentioned in Section 2.3, the security demands at NTNU are not as high as in national, political elections. The contents of individual votes should remain safe at least 5 years forward in time. The special purpose machine proposed by Bernstein in [4], may be used by an adversary for efficiently breaking the security of a system, based on the factorization of large integers. In light of this special purpose machine, it may seem that the length of the keys in public key algorithms, based on the factorization of large integers, should be very long in order to preserve the security of the system. However, the construction of such a machine is likely to be expensive, and it is assumed to be unlikely that anyone will go to such measures to break the encryption of the votes in an election at NTNU. The length of the keys chosen in a cryptosystem will always be a trade-off between security and performance. As this system is designed to be universally verifiable for any participant in an election, the time spent on this verification must be kept as low as possible. Based on the results of a workshop held by NIST in 2001 [14] and recommendations from RSA Security [21], it is assumed that a key length of 1024 bits should be sufficient to secure the votes for long enough time for elections at

NTNU. Other elections with higher security requirements may choose a larger key length to obtain a higher level of security, with some loss of performance.

The threshold variant of the cryptosystem proposed in [19] will be used to secure the *privacy* of the votes. The homomorphic property of this cryptosystem enables the encryptions of all valid votes to be combined into an encryption of the result, by multiplying the encrypted votes together. The threshold variant of the cryptosystem enables the results to be decrypted without allowing a single entity the possibility of learning how single voters voted.

Votes posted on  $B$  will be associated with the username of the voter. The ID of the voter is also included in the input to the hash function  $H$ , when creating the proof of correctness of the vote. This ensures that no voter can post two votes on  $B$  with the same ID. However, the property *no double voting* cannot be guaranteed without some kind of access control of the system. This is not handled by this voting system. It may be handled by the internal network authentication at NTNU.

The votes and computations made by the decryption servers will be secured using zero-knowledge proofs. As explained in Section 3.4, these proofs have a security parameter  $q$ , which is the bitlength of the challenge provided by  $V$  in the interactive variant, and the bitlength of the digest of  $H$  in the non-interactive variant. With a  $q$  of 80 bits, an incorrectly formed vote will be accepted with a probability of  $\leq 2^{-80}$ . This probability is assumed to be small enough to ensure that the requirements *no cheating* and *correctness* will be satisfied for this system.

All zero-knowledge proofs posted on  $B$  may be verified by any participant in the election. This ensures that anyone may verify all votes posted on  $B$ , as well as the proofs proving the correctness of the calculations carried out by the decryption servers, thereby making the elections universally *verifiable*.

Innsida is a secure internal network at NTNU, which authenticates users by their username and password. As Innsida will be used to control  $B$ , this will be the only part of the system which needs to be on-line during the whole election, thus the voting system does not satisfy the property *off-line*. This part of the system must be able to withstand denial of service attacks. Innsida is maintained by NTNU and will be continuously monitored by the NTNU staff during the election. Multiple instances of  $B$  may be launched if there is need for it, using existing server replication techniques. The *robustness* of the system will largely depend on the implementation and integration of the system, and cannot be guaranteed by the system itself.

This system will not satisfy receipt-freeness. Any voter can easily post the random numbers used in the encryption of votes to show anyone the contents of the votes. As vote-buying and coercion are minor concerns at NTNU, the lack of this property is not considered critical.

The requirements satisfied by this voting system are shown in Table 4.2. As we can see by comparing this table to Table 2.1, this system may be implemented to satisfy all properties required in an NTNU election.

Privacy	X
No Double Voting	(X)
No Cheating	X
Correctness	X
Verifiable	X
Off-line	
Receipt-Freeness	
Robustness	(X)
Ranked Voting	X
Internet Voting	X
Usable in a heterogeneous environment	X

Table 4.2: Requirements satisfied by the voting system proposed in this thesis.

The requirements in parenthesis cannot be guaranteed by the system alone, and will depend on the implementation of the system and the access control enforced when using the system for an election.

## 4.7 Other Considerations

While evaluation of security measures other than those concerning the cryptographic protocols is beyond the scope of this thesis, this section will describe some challenges regarding the integration of this system at NTNU.

The security of the internal network at NTNU, Innsida, will be the weakest link of this voting system, if used for authentication of voters and access control of the voting system. Even though NTNU has a strict policy concerning the quality of the Innsida passwords (it has to consist of 8 characters, and include at least one character from each of the groups a-z, A-Z and 0-9), it may be possible for an adversary to gain access using another person's credentials. It is not uncommon that students and employees leave computers unattended, without logging out of Innsida. Also,

many employees at NTNU have access to the authentication mechanisms used, and the possibility of employees posting votes for other people or even deleting votes from *B* cannot be ruled out.

No PKI<sup>5</sup> is in place at NTNU as of this moment, thus there is no straight-forward way to create a digital signature on the information posted on *B*. This implies that there is no way to secure the integrity of votes posted on *B* cryptographically. One possible way of securing that votes stay unaltered on *B* is to give the voter a receipt with the vote and proof, so the voter can verify the information on *B* at the end of the election. However, this cannot guarantee the authenticity of all information posted on *B*.

Another problem is the key distribution. Who can be trusted to generate the keys for the voting system, and how will the keys be delivered to the servers and the voters? These are questions that need to be answered in order to realize this voting system securely for voting at NTNU.

## 4.8 Summary

This chapter has explained the different components of the voting system, and how they will work together for carrying out a secure election. The voting methods of this system were explained, as well as their performance. A description of how this voting system will satisfy the requirements for voting at NTNU was also provided. Some security issues that need to be solved for attaining a high level of security for voting at NTNU were raised at the end of this chapter. The next chapter will cover the implementation of the system.

---

<sup>5</sup>Public Key Infrastructure. In cryptography, a public key infrastructure (PKI) is an arrangement that provides for third-party vetting of, and vouching for, user identities. It also allows binding of public keys to users.

## Chapter 5

# Implementation

This chapter will discuss the implementation of the voting system. This implementation is mainly done for performance evaluation of the algorithms, thus the security of the implementation is not guaranteed.

### 5.1 Overview

The programming language of choice is Java, as it functions well in a heterogeneous environment and has good mechanisms for implementing a distributed system. Java has a built-in class called *BigInteger*, which is capable of performing computations using large integers. This class is needed in order to implement the cryptosystem. The implementation supports different voting methods: elections for 1-out-of- $L$  candidates and elections for  $t$ -out-of- $L$  candidates. Both of these methods are implemented using two different encodings of the votes, which perform differently depending on the parameters of the election. The two encodings differ largely in the techniques needed to create and validate proofs, as well as in the number of proofs needed as explained in Section 4.5. A Yes/No-election can easily be obtained by using the 1-out-of- $L$  method with  $L = 2$ .

The implementation of the system is realized as a distributed system using Java RMI<sup>1</sup>. The system consists of four different components: The voter application, the decryption server, the election authority and the bulletin board.

---

<sup>1</sup>Java Remote Method Invocation. For more information on Java RMI see [11]

The voter application does not provide any authentication and access control, as this is out of scope for the thesis, but this may easily be provided by introducing a layer providing these services, below the RMI layer in the model, as described in Figure 4.2.

The deployment diagram of the system is shown in Figure 5.1. The remote object *BulletinBoard* handles the storage of votes and decryption shares during the election process. This is the only entity that needs to be on-line during the whole duration of the election. The *ElectionServer* is bound to the registry by the *ElectionAuthority* and provides the voters and servers with the public key of the election, as well as the election parameters. The *ElectionServer* also provides the *DecryptionServers* with their key shares, which will be used in the share decryption process.

All entities in the system share a common library of resources in addition to these shared objects. This library contains the classes which hold the different types of votes, the algorithms needed to verify the zero-knowledge proofs used throughout the system, as well as the classes holding the public key for the election and the key shares used by the decryption servers. The remote interfaces needed for using the remote objects through Java RMI are also included in this library.

## 5.2 Implementation of the Cryptosystem

The cryptosystem used in this voting system is a threshold variant of the generalization of Paillier's public-key system. The cryptosystem mainly consists of four parts: the key generation, the encryption, the share decryption and the share combination.

The random key generator used in this implementation is provided by the generic *SecureRandom* class included in Java. *SecureRandom* is a cryptographically strong random number generator, satisfying the criteria stated in the recommendation in [7], and should satisfy the requirements needed for randomness in this cryptosystem.

Key generation for a non-threshold variant of this cryptosystem is fairly straightforward, but in the case of the threshold variant there are more factors to consider. As explained in Section 3.6.1, we need to find two primes  $p$  and  $q$ , that satisfy  $p = 2p' + 1$  and  $q = 2q' + 1$ , where  $p'$  and  $q'$  are primes and different from

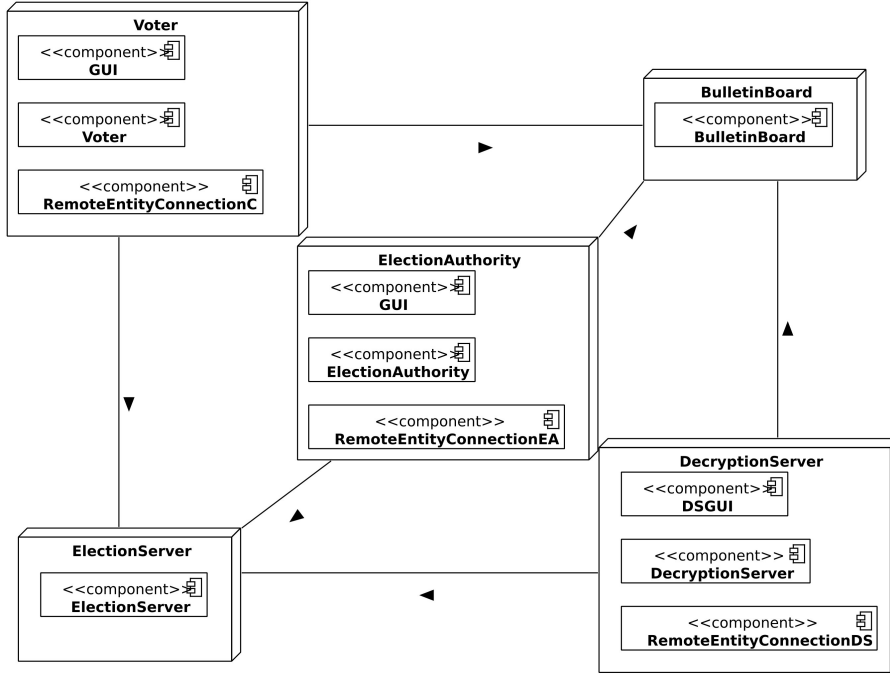


Figure 5.1: Deployment diagram of the voting system. The two remote objects are the *BulletinBoard* and the *ElectionServer*. The use of these remote objects is shown by connectors in the diagram.

$p$  and  $q$ . In this implementation the key generation is performed by trying and failing until a suitable pair of primes is found. The *BigInteger* class includes an implementation of the Miller-Rabin test, which may be used to determine whether a number is a prime or not, with a given certainty. This is used to determine whether the values chosen for  $p, p', q, q'$  are prime or not. In this implementation of the key generation, the probability of error, or uncertainty, is set to be  $0.5^{100}$ .

After finding suitable primes, the parameters  $n$  and  $m$  are calculated as  $n = pq$  and  $m = p'q'$ . A suitable secret key  $d$  is chosen to satisfy  $d = 0 \pmod m$  and  $d = 1 \pmod n^s$ , by using the Chinese remainder theorem. The key shares for the decryption servers are computed as explained in Section 3.6.1.

Encryption of a given plaintext  $i$  is done by choosing a random  $r \in \mathbb{Z}_n^*$ , and computing the ciphertext as explained in Section 3.2.2. A discussion of optimizations regarding this computation follows in Section 5.5.2.

The share decryption process is performed by raising the ciphertext to the power of the decryption server's own secret exponent. Along with this exponentiation the server includes the zero-knowledge proof for convincing a verifier that  $\log_c^4(c_i^2) =$

$\log_v(v_i)$ . This is done non-interactively using a hashfunction, and provides security in the random oracle model.

The combination of shares may be done without any knowledge of the secret keys. When enough shares are acquired, the procedure explained in Section 3.6.4 may be applied to yield the plaintext.

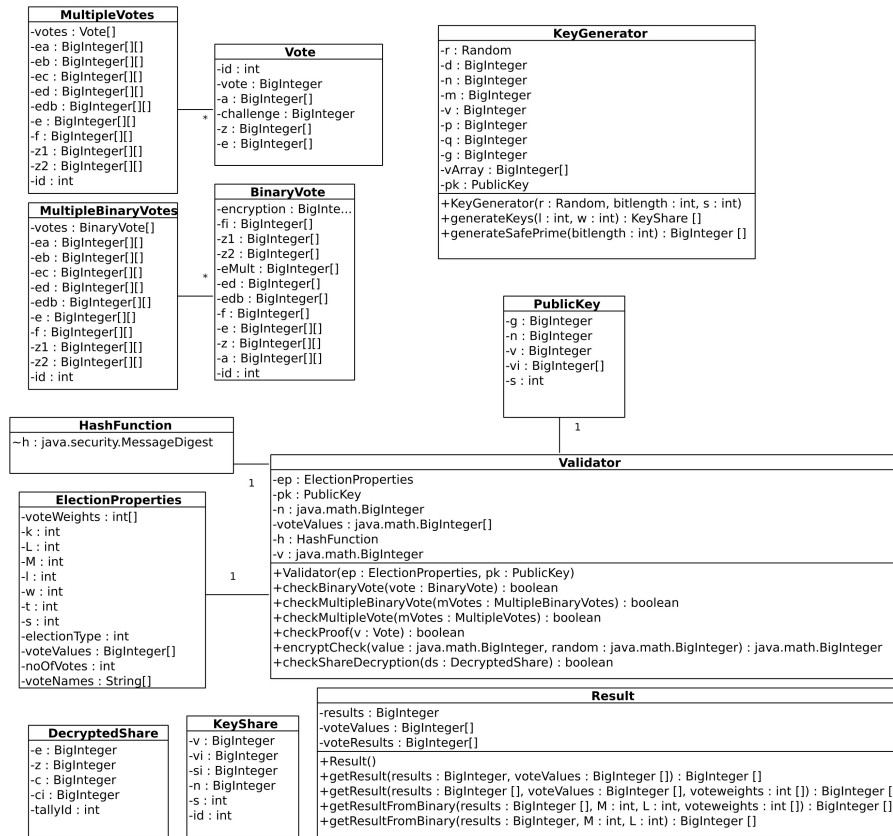
### 5.3 Implementation of the Common Library

All of the components of the voting system need a shared library of resources, which specifies the data structures for the different types of votes and keys. In addition to these data structures, classes that are used by multiple components in the system are included in this common library. The package containing these classes is called *votingSystem*, and the class diagram for this package is shown in Figure 5.3. In addition to this package, the packages *remoteInterfaces* and *exceptions* are needed. The *remoteInterfaces* package simply contains the interfaces needed to interpret the remote objects placed in the RMI registry. The *exceptions* package contains the exceptions that may be thrown if any problems occur in the voting process.

The classes *Vote*, *MultipleVote*, *BinaryVote* and *MultipleBinaryVote* are the data structures holding the different types of votes, as well as their proofs of correctness. The attributes of these classes should be easily interpreted after reading Section 4.5, which explains the different voting methods used, as well as Section 3.4, which explains the proofs used in the methods. Note that *MultipleVote* is a class holding the multiple *Vote*-objects in the case of a multi-vote election, as well as the additional attributes needed to prove that all votes are cast for different candidates. This is also the case for *MultipleBinaryVote* and *BinaryVote*.

The *PublicKey* and *KeyShare* classes are the data structures holding the keys used in the election. The *PublicKey* class specifies  $n$ ,  $g$ , and  $s$ , which are needed to encrypt votes for a specific election. Additionally the class specifies the public parameter  $v$ , and the list of verification keys  $vi$ , which are needed for verification of the calculations made by the decryption servers. The *KeyShare* class specifies the secret exponent  $si$ , used in the share decryption process, along with the public parameter  $v$  and the verification key  $vi$ , as explained in Section 3.6.3.

The *DecryptedShare* class is the data structure holding a decrypted share, along

Figure 5.2: Class diagram of the package *votingSystem* including only the most important methods and attributes.

with the proof which will convince a verifier that the server indeed raised the ciphertext to the power of his secret key *si*.

The *ElectionProperties* class is the data structure holding the parameters of the election, which are specified by the election authority.

All the classes mentioned above are passed between components using Java RMI. Therefore these classes implement the interface *Serializable*, which makes it possible to pass them as arguments to and from a remote object.

The *HashFunction* class provides the standard Java implementation of the hash function SHA-256, which is used in all the zero-knowledge proofs in this system. SHA-256 gives message digests up to 256 bits, which is the maximum value for the security parameter *q* in this system. Because the *HashFunction* class truncates the output from the hash function on byte level, *q* can only be chosen as a multiple

of 8 in this system.

The *Validator* class includes all the methods needed to validate the zero-knowledge proofs used in this system. The votes cast and the shares acquired from decryption servers may therefore be validated by anyone participating in the election.

The *Result* class includes methods for extracting the result of the election from the plaintext. The procedure for finding the result depends on the encoding used for the votes. This class includes four methods, and the method used must correspond with the voting method used in the election.

## 5.4 Implementation of the Election Authority

The election authority is located in the *ea* package of the voting system. This section describes the implementation and functionality of this component.

### 5.4.1 Description of the Structure

The election authority generates the keys and sets the parameters of the election. In this implementation, the election authority also handles the combination of shares for obtaining the final result, although this is a process which may be executed by anyone, after enough shares are acquired. The class diagram in Figure 5.3 shows the structure of the election authority. The *GUI* collects the election parameters from the election administrator and launches the *KeyGenerator* to generate the public key and the key shares for the decryption servers. This information is stored in the remote *ElectionServer* object, which will be bound to the RMI registry through the *RemoteEntityConnectionEA* object. For simplicity the RMI registry runs on the election authority component in this implementation, but it could easily be run separately on a remote server.

The bulletin board could be started independently of the election authority, but since the RMI registry is needed for the bulletin board to register, the bulletin board cannot be started until the election authority is started, in this implementation.

The sequence diagram in Figure 5.4 describes the actions of the election authority through the whole duration of an election. The application starts by receiving the election parameters from the election administrator. The application then starts the *KeyGenerator* for generation of the public key and the key shares (not shown in

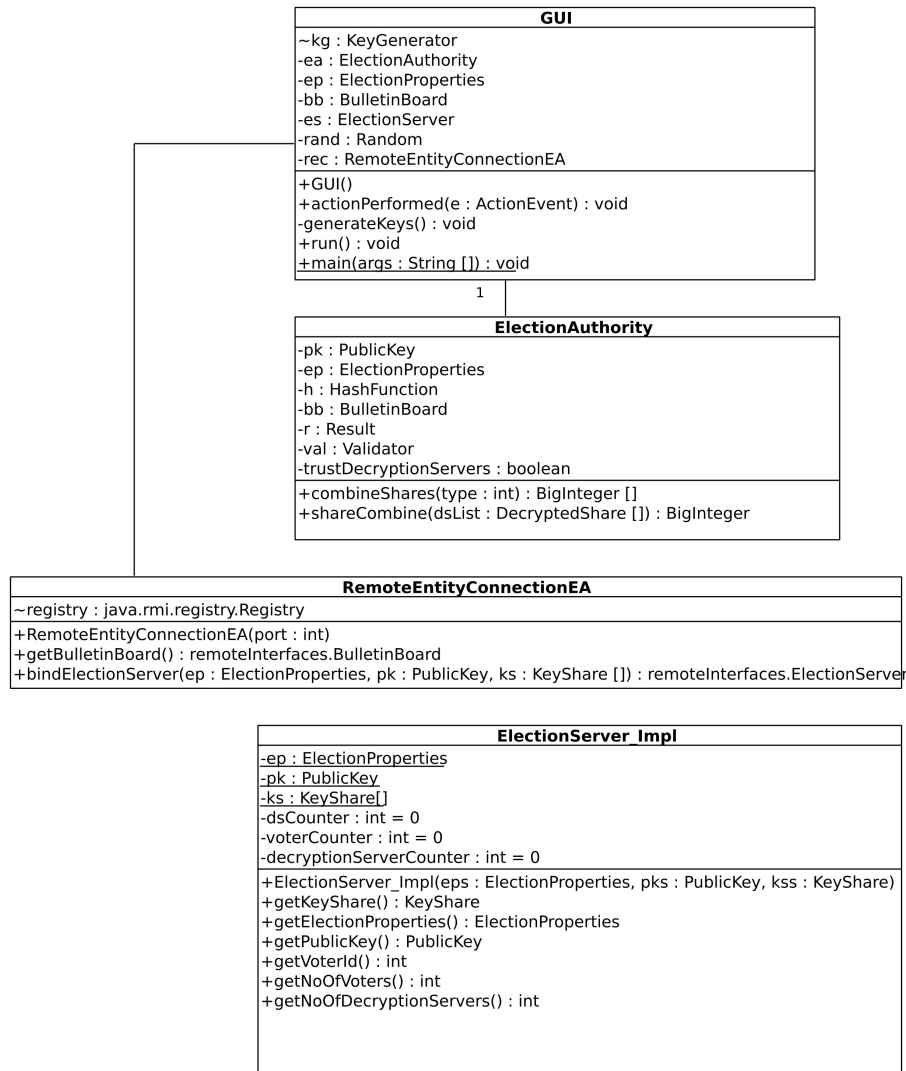


Figure 5.3: Class diagram of the *ea* package including only the most important methods and attributes. The *ElectionServer\_Impl* class is the implementation of the remote object that will be bound in the RMI registry for remote access.

Figure 5.4). *RemoteEntityConnectionEA* is then instantiated and used for starting a *RMIRegistry* where remote objects may be stored. The *ElectionServer* object is bound to the *RMIRegistry*, and the election is started. The remote object *BulletinBoard* may now be bound to the *RMIRegistry*, and voters and decryption servers may use the *ElectionServer* object to retrieve the keys and the election parameters. The voters may now post their votes to the *BulletinBoard*.

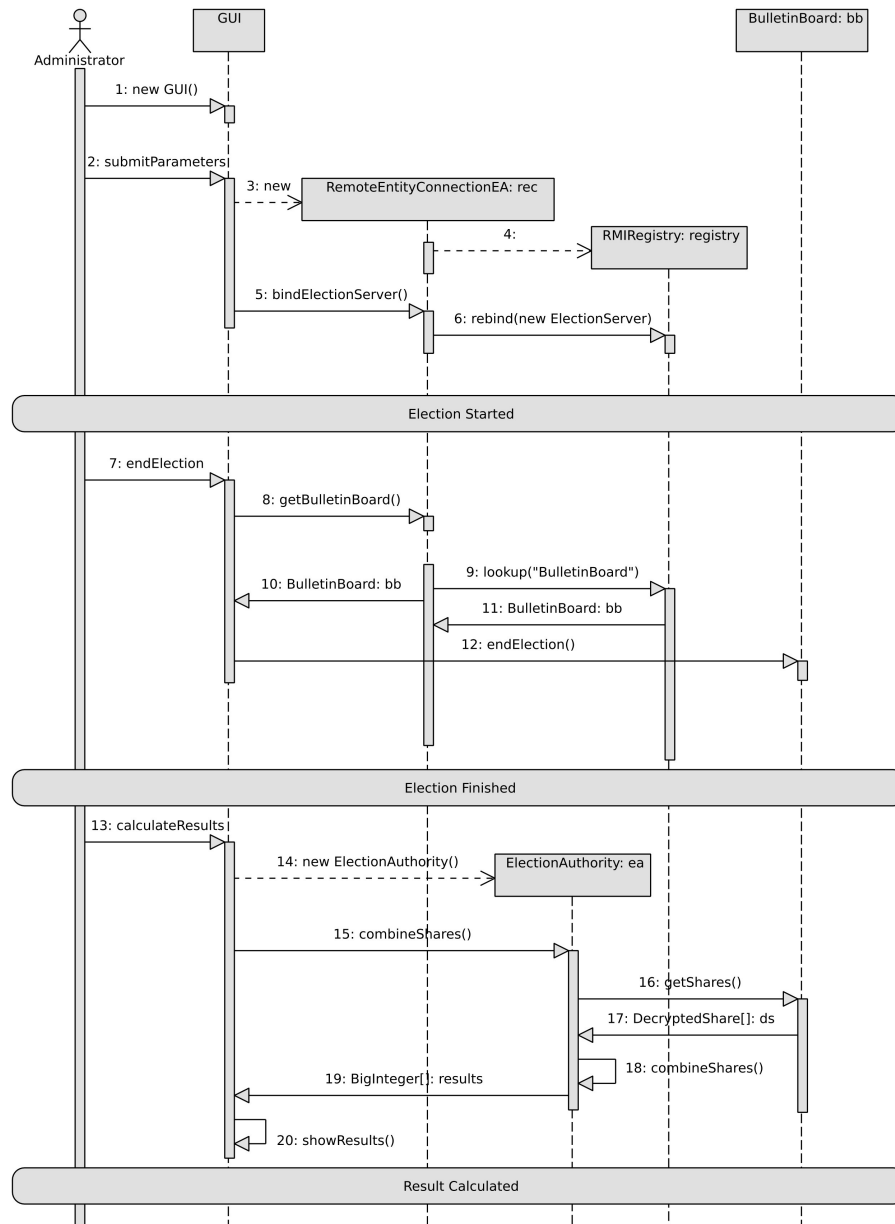


Figure 5.4: The sequence diagram for the election authority

The administrator may end the election at any time. This is done by retrieving the *BulletinBoard* stub through the *RemoteEntityConnectionEA*, which will look up the *BulletinBoard* in the *RMIRRegistry* and return the stub. The method *endElection()* is then invoked at the *BulletinBoard*. This will cause the *BulletinBoard* to stop

accepting votes, and notify the decryption servers that share decryption of the result may start.

The election authority monitors the number of shares posted to the *BulletinBoard* by the decryption servers. These shares are calculated by the decryption servers and posted at the *BulletinBoard*. When enough shares are acquired, these shares may be combined to yield the result of the election. This is done by retrieving the shares from the *BulletinBoard*, and instantiating the *ElectionAuthority* class which will check the proofs of correctness and return the result.

#### 5.4.2 Parameters Chosen by the Election Authority

All the parameters shown in Table 4.1 are chosen by the election authority, except  $s$ , which will be calculated automatically to adjust the message space, and the generator  $g$ , which is fixed to  $(1 + n)$  in this implementation. In addition to these parameters the administrator of the election inputs the type of election, the weights of the votes in the case of a multi-vote election, and the names of the different candidates. A screenshot of one of the two administration windows is included in Figure 5.5.

### 5.5 Implementation of the Voter Application

This section describes the general structure of the voter application, as well as some of the optimizations done to improve the performance of the encryption. The package containing the voter application is called *voter*.

#### 5.5.1 Description of the Structure

The *voter* package is implemented as a Java application. The application can easily be ported to an applet for use in Internet elections. It consists of three classes: *GUI*, *Voter* and *RemoteEntityConnectionC*. The *GUI* specifies the user interface and receives input from the voter. The *Voter* class encrypts the votes using the correct encoding for the election type, and creates the proofs of correctness. The *RemoteEntityConnectionC* is used to obtain the stubs of the remote objects from the *RMIRRegistry*. These stubs are used to obtain the parameters of the election, as well as the public key. The stubs are also used by the *Voter* class to send the

The screenshot shows a window titled "Election Authority <2>". Inside, there is a section "Set up the election (1/2)". Below this, there are several labeled input fields, each with a numeric spinner control:

- Keylength of Public Key: 1,024
- Maximum number of voters (M): 1,000
- Number of votes (V): 2
- Security parameter. Length of bitchallenges (Q): 80
- Number of candidates (L): 6
- Number of decryption servers (S): 10
- Number of decryption shares needed (W): 5
- Type of election: 1-out-of-L (dropdown menu)

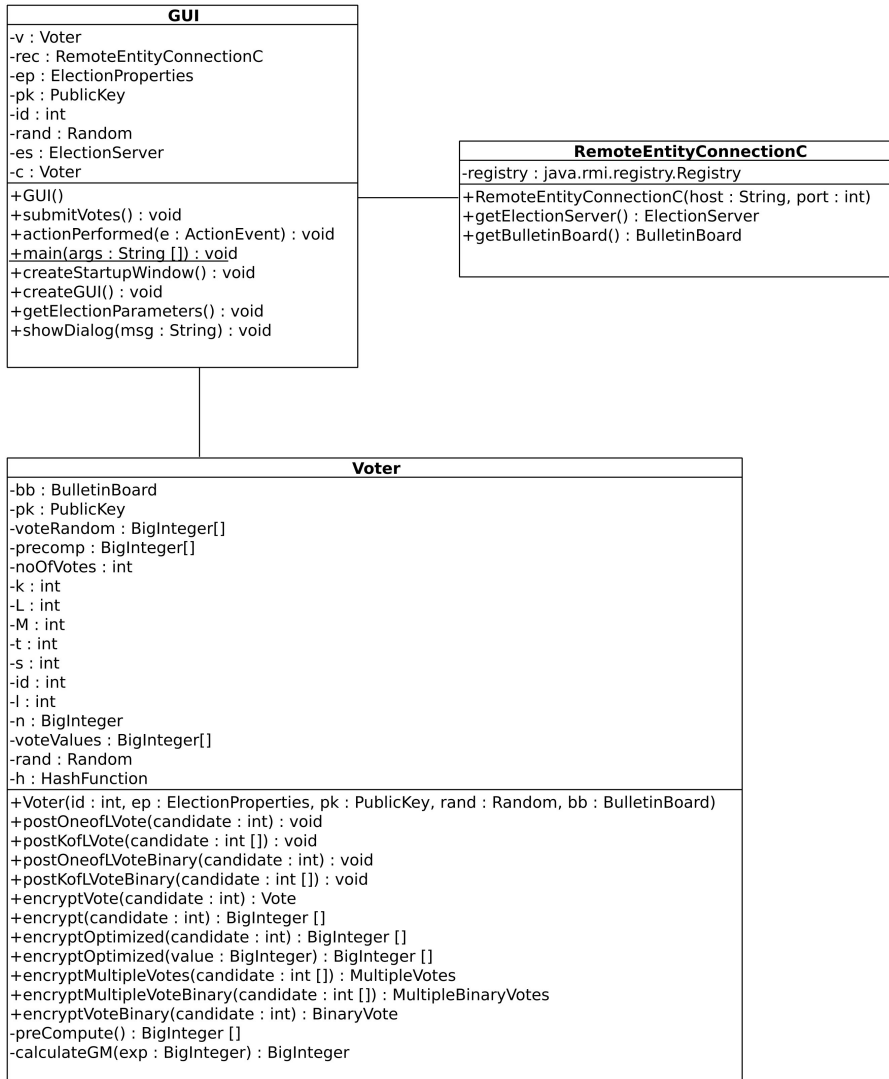
At the bottom of the window, there is a button labeled "Next Page".

Figure 5.5: Screenshot of the administration window at the election authority.

encrypted vote to the bulletin board. If any kind of problem occurs when trying to post votes, the application notifies the voter, and the voter may try again later. This will happen if either the *RMRegistry* is off-line, or the *BulletinBoard* object is not yet bound to the *RMRegistry*.

As specified in the class diagram shown in Figure 5.6, the *Voter* class includes four different methods for encrypting votes, depending on the type of election and the encoding of votes. The election type is retrieved together with the other parameters of the election, through the use of the remote *ElectionServer* object. The sequence diagram for the total lifetime of the voter application can be seen in Figure 5.7.

When the voter starts the application he is prompted for the hostname and the port number used by the *RMRegistry*. The program then retrieves the stubs for the remote objects *ElectionServer* and *BulletinBoard* through the *RemoteEntityConnectionC*. The remote *ElectionServer* is used for retrieving the *PublicKey* and the *ElectionProperties* for the system. The *GUI* then creates a *Voter* object for performing the computations. The choices of the user is collected, and the vote as well as the proofs of correctness are inserted into the proper container for the

Figure 5.6: Class diagram of the package *voter*, including only the most important methods and attributes.

votes. The *postVote()* method corresponding to the current election type in *Voter* is invoked. Finally, the *Voter* object sends the encrypted vote and the proofs to the *BulletinBoard* using the stub obtained from *RemoteEntityConnectionC*.

The different methods used for encryption of votes and creation of proofs for different election types may be seen in Section A.1.

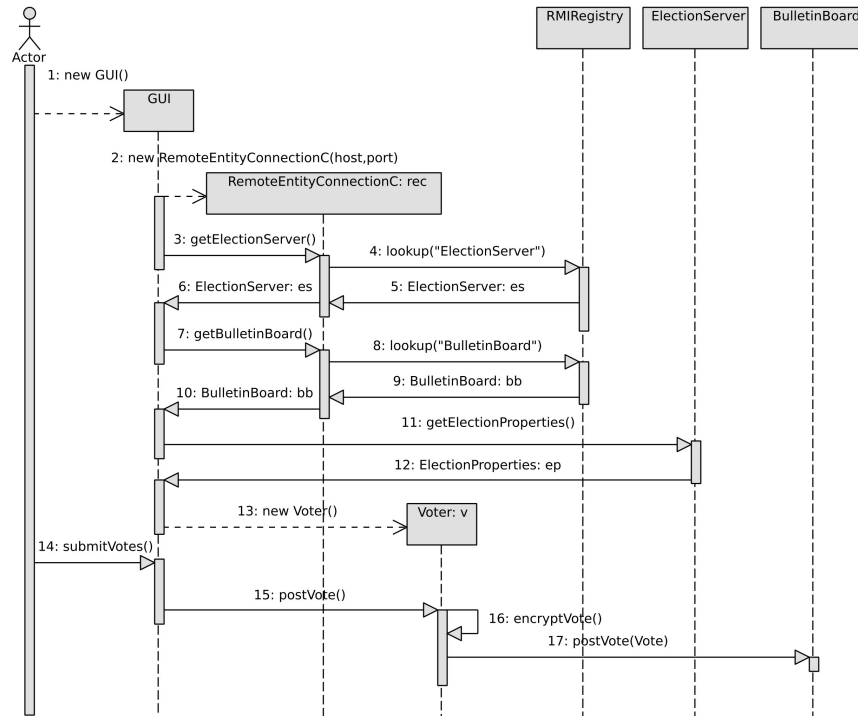


Figure 5.7: Sequence diagram for the voter application.

### 5.5.2 Optimizations of Encryption

The encryption algorithm is used for encryption of votes as well as the creation of zero-knowledge proofs. This is explained in 3.2.2. The large exponentiations performed in the encryption are time-consuming and should be optimized in order to make the voting system more efficient. Calculating the expression  $(1 + n)^m$  directly gives an exponentiation with a  $O(n^s)$  exponent. The source code for this solution is shown below:

```

BigInteger temp1 = pk.getG().modPow(voteValues[candidate], nArray[s + 1]);
2 BigInteger temp2 = v.modPow(nArray[s], nArray[s + 1]);
BigInteger ciphertext_vote = (temp1.multiply(temp2)).mod(nArray[s + 1]);

```

Listing 5.1: Unoptimized encryption algorithm

Even though the generator  $g$  in the voting system may be chosen from many possible values, it is efficient to choose  $g = 1 + n$ . This may be done without any

reduction of security. As explained by Jurik in [10] we note that:

$$(1 + n)^m = 1 + \binom{m}{1}n + \cdots + \binom{m}{s}n^s \mod n^{s+1}$$

Using this, all the binomial terms may be calculated separately, which gives  $O(s)$  multiplications, using:

$$\binom{m}{j} = \binom{m}{j-1} \frac{m-j+1}{j}$$

By precomputing the values which are constant for every encryption we can optimize this further. All exponentiations of  $n^i$  for  $i \in (1 \text{ to } s)$  are also precomputed. We precompute  $(j!)^{-1}n^j$  for every  $j$  as shown in the Java code below:

```

BigInteger precomp[] = new BigInteger[s+1];
2 BigInteger jFac = BigInteger.ONE;
  precomp[1] = n;
4 for(int j=1 ; j<=s ; j++){
    jFac = jFac.multiply(BigInteger.valueOf(j));
6    precomp[j] = jFac.modInverse(nArray[s+1]);
    precomp[j] = precomp[j].multiply(n.pow(j)).mod(nArray[s+1]);
8 }

```

Listing 5.2: Calculation of the precomputed values

The full code for computing  $(1 + n)^m$  when using the precomputed values is shown in the listing below:

```

BigInteger c = BigInteger.ONE.add(voteValue.multiply(n));
2 BigInteger temp = voteValue;
  for(int j=2 ; j<=s ; j++){
4    BigInteger j_ = BigInteger.valueOf(j);
    temp = temp.multiply((voteValue.subtract(j_)).add(BigInteger.ONE)).mod(nArray[s-j+1]);
6    c = (c.add(temp.multiply(precomp[j]))).mod(nArray[s+1]);
  }

```

Listing 5.3: Calculation of  $(1 + n)^m$

An analysis of different methods of computing  $(1 + n)^m$ , as well as the computational gain of this optimization is given in [10].

## 5.6 Implementation of the Decryption Server

The purpose of the decryption server is to retrieve the encrypted votes from the server, multiply the valid votes together, and then make a decrypted share of the

result, along with a zero-knowledge proof, which proves the correctness of the calculations. The decryption server is located in the *ds* package of the voting system.

### 5.6.1 Description of the Structure

The structure of the decryption server can be seen in Figure 5.8. The *ds* package contains three classes: *DSGUI*, *RemoteEntityConnectionDS* and *Server*. The *DSGUI* provides the user interface and takes input from the administrator of the decryption server. The *Server* performs the calculations for the share decryption of the result. *RemoteEntityConnectionDS* is responsible for retrieving the stubs of the remote objects needed by the decryption server. The user interface of this component allows the administrator of the decryption server to check the status of the election, and to see the number of votes cast. When the election has finished, the decryption server is allowed to retrieve the votes from the bulletin board, perform the share decryption, and post the decrypted share on the bulletin board.

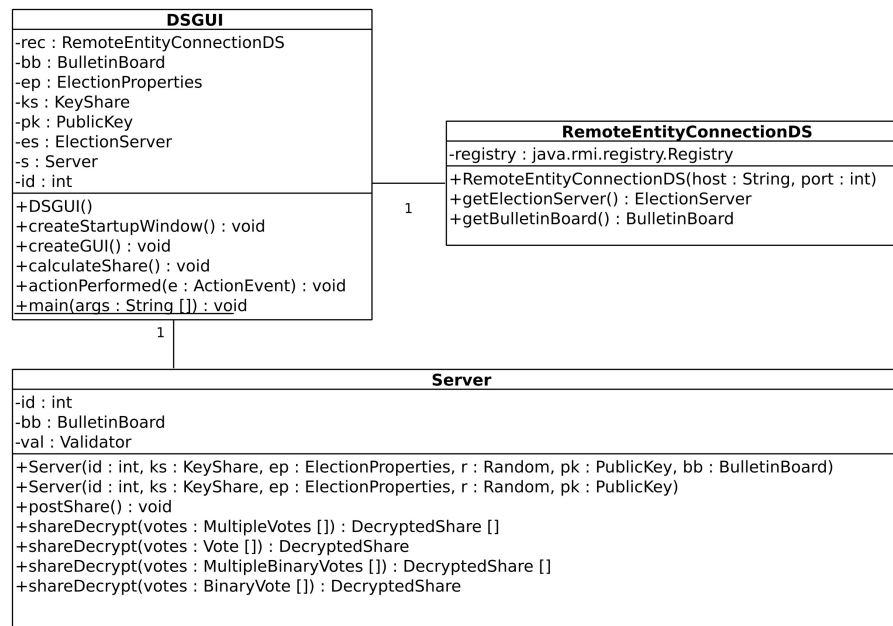
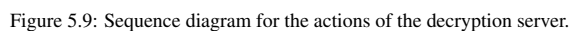


Figure 5.8: Class diagram of the *ds* package, including only the most important methods and attributes.

The sequence diagram for the actions of the decryption server throughout the election is shown in Figure 5.9. For simplicity, some of the objects used by the decryption server for validation of votes are excluded from the diagram.



When the decryption server starts up, it connects to the *RMIRegistry* using the *RemoteEntityConnectionDS* and retrieves the stub for accessing the *ElectionServer* object. The election parameters, the public key for the election and the secret key share are then retrieved by calling the appropriate methods on this stub. When the election is finished, the decryption server administrator is notified through the user interface. The share decryption process is started on input from the server administrator. The stub of the *BulletinBoard* is retrieved using the *RemoteEntityConnectionDS*. The stub of the *BulletinBoard* is then passed on to the *Server*, which retrieves the votes using the *BulletinBoard* stub. The *Server* verifies the correctness of the votes, multiplies the valid votes, and creates a proof of correctness. The result of the exponentiation and the proof of correctness is finally sent in the

form of a *DecryptedShare* to the *BulletinBoard*.

### 5.6.2 Optimizations of the Decryption Server

The validation of votes requires many time-consuming computations, especially when dealing with multi-vote elections. As described in Section 4.5, the number of zero-knowledge proofs needed for proving correctness of votes grows dramatically as the number of votes per voter increases.

As an example, in a  $t$ -out-of- $L$ -election with  $t = 4$  and  $L = 10$ , the decryption server needs to validate the 4 votes, verifying that each of these votes encrypt one of the 10 plaintexts allowed using the 1-out-of- $L$   $n^s$ 'th power protocol. In addition to this verification, the cross-validation of votes proving that every pairwise difference of votes is non-zero, requires  $\frac{t^2}{2} - \frac{t}{2}$  multiplication-mod- $n^s$  proofs. In this case we are required to verify 6 proofs for the cross validation. When considering that this process needs to be repeated for each voter, the computations must be done as efficiently as possible, in order for this voting system to be useful.

To minimize the number of computations done in validation of proofs, some frequently used values are precomputed. The validation of multiplication-mod- $n^s$  proofs requires the server to perform encryptions. These encryptions are optimized as explained in Section 5.5.2. When verifying a *MultipleBinaryVote* or a *BinaryVote*, the value  $l + 1$ , which defines the bitlength of  $L$ , and the values  $(g^{M^{2^i}})^{-1}$  for  $0 \leq i \leq l$ , are precomputed.

In addition to these optimizations, the values for the exponentiation  $n^i$ , for  $0 < i \leq s + 1$ , are precomputed and accessed from an array. The number of objects used in the implementation of critical methods is also kept to a minimum for optimizing performance. The Java implementation of the methods used for validation of the zero-knowledge proofs is included in Section A.3

## 5.7 Implementation of the Bulletin Board

The implementation of the bulletin board is located in the package *bb*. It contains the classes: *BulletinBoardServer*, *BulletinBoard\_Impl* and *RemoteEntityConnectionBB*. The *BulletinBoardServer* specifies the user interface of the bulletin board. *BulletinBoard\_Impl* is the implementation of the remote object *BulletinBoard*,

which will be bound in the *RMIRegistry*. The *RemoteEntityConnectionBB* is the class responsible for the interaction with the *RMIRegistry*.

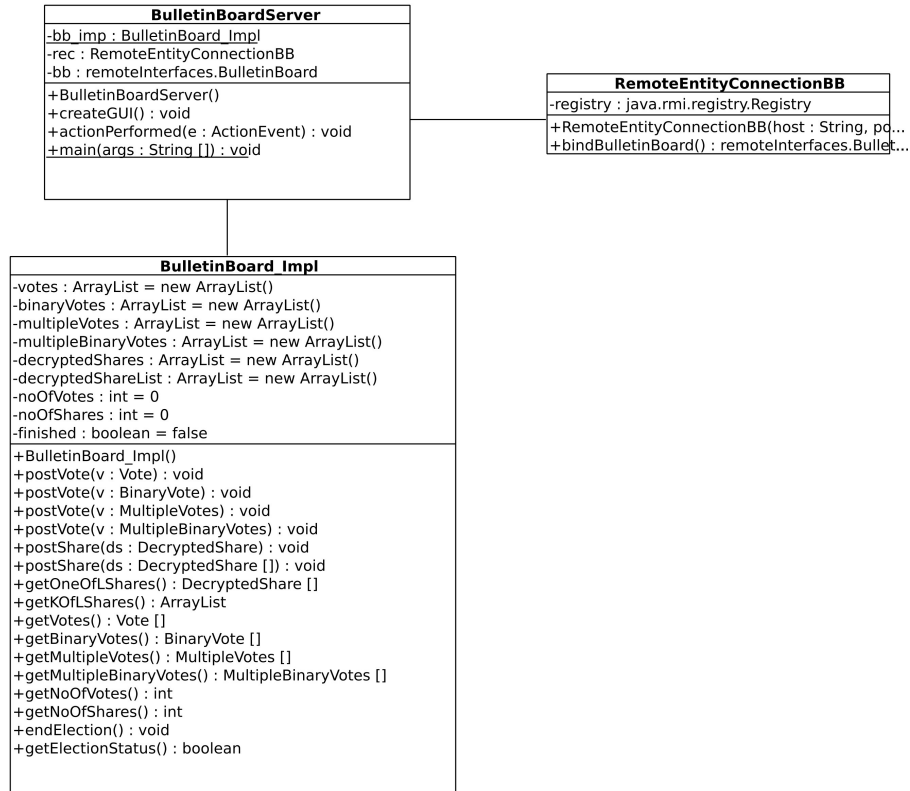


Figure 5.10: Class diagram for the package *bb* containing the bulletin board.

The implementation of the remote object *BulletinBoard* is a remote data structure holding votes and decrypted shares (including the proofs associated with these elements). In addition to holding votes and shares, the *BulletinBoard* provides real-time information of the election, showing the number of votes given, the number of decrypted shares posted and the status of the election.

The *BulletinBoard* has no interaction with the *ElectionServer* and does not have any knowledge of the parameters of the election. This implies that the *BulletinBoard* allows any voter to post any type of vote at any time. This should, however, never be an issue because every participant of the election should know what election type is used from the *ElectionProperties* obtained from the *ElectionServer*, and incorrect votes should not be counted by the decryption servers. If a decryption server counts incorrect votes, this will be discovered when verifying the share

decryptions.

Due to the simplicity of this package, no sequence diagram will be included here. The functionality of this package may easily be interpreted from the class diagram shown in Figure 5.10.

## 5.8 The Simulation Tool

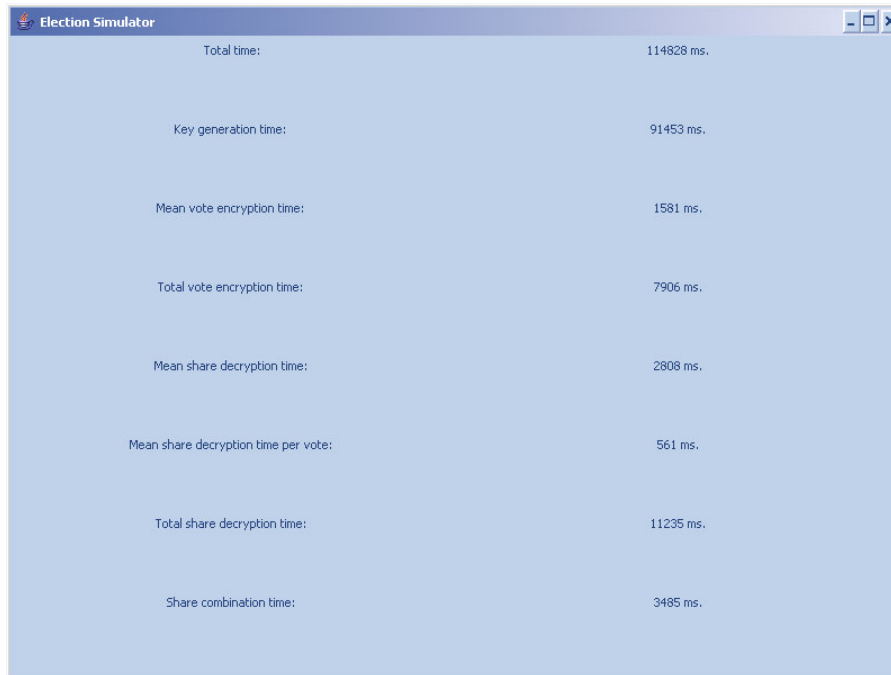
In addition to the distributed voting system, a package called *simulator* is included. This package is included for simulating elections. Simulations are important to evaluate the performance of the system in different scenarios. The simulation tool needs all packages to be present, as it runs all components of the election locally.

The simulator includes a user interface much like the one in the *ea* package, where the user is allowed to choose the parameters of an election. In addition to the parameters, the user is allowed to choose how many simulated votes that will be cast.

The simulator does not run as a distributed system, but performs all computations locally. This implies that networking delays will not be simulated using this tool. However, this should not be a major concern, since the network delays of RMI should be insignificant compared to the time spent on computations in this system, due to the small amount of data transferred between components. This is especially true when considering the high data rate of most Internet connections today.

The times displayed to the user after a simulated election are: total time, key generation time, mean vote encryption time (the mean time spent by a voter for encryption of votes and creation of proofs), total vote encryption time, mean share decryption time (the mean share decryption time for all decryption servers, which includes the validation of all votes and the share decryption of the result), mean share decryption time per vote cast, total share decryption time and share combination time. An example of the results of a simulation can be seen in Figure 5.11.

The *simulator* package contains only one class, which is called *Simulator*. *Simulator* contains the user interface, as well as the logic for simulating an election. The simulation tool can be executed by running the command "`java simulator.Simulator`", or by running the shell script *simulator.sh* in Linux or *simulator.bat* in Windows.

A screenshot of a Java window titled "Election Simulator". The window has a light blue background and a standard Windows-style title bar with minimize, maximize, and close buttons. It displays a list of performance metrics for an election simulation. The metrics are arranged in two columns: the metric name on the left and the time value on the right.

Total time:	114828 ms.
Key generation time:	91453 ms.
Mean vote encryption time:	1581 ms.
Total vote encryption time:	7906 ms.
Mean share decryption time:	2808 ms.
Mean share decryption time per vote:	561 ms.
Total share decryption time:	11235 ms.
Share combination time:	3485 ms.

Figure 5.11: Screenshot of the simulation results

## 5.9 Executing the Voting System

This section will function as a tutorial for running an election using the voting system.

The bulletin board should be executed from the same machine as the election authority in this implementation, as binding an object to the RMI registry from a non-local host is restricted in Java RMI.

The Java packages for all components of the system are located in the *Election-System* folder of the digital appendix included with this thesis. All contents of this appendix is described in Appendix B. The programs require that Java Runtime Environment<sup>2</sup> is installed. The voting system should work on all operating systems capable of running Java Runtime Environment.

**Starting the Election Authority.** The election authority can be started by running the shell script *ea.sh* in Linux and *ea.bat* in Windows. The script takes the

---

<sup>2</sup>Available from <http://www.java.sun.com>

port number for the RMI registry as argument. If no port is given, the program uses the default port 2004.

**Starting the Bulletin Board.** The bulletin board is started by running the shell script *bb.sh* in Linux and *bb.bat* in Windows.

**Starting the Decryption Server.** The decryption server is started by running the shell script *ds.sh* in Linux and *ds.bat* in Windows.

**Starting the Voter Application.** The voter application is started by running the shell script *voter.sh* in Linux and *voter.bat* in Windows.

All of these components require RMI support. To secure that the applications are allowed to communicate over the network, a policy file is included for use with the applications. This file is used by all these components and is included as *policy*. This file must be present in the ElectionSystem directory.

All components require the user to input the hostname/IP-address and the port number of the the machine running the election authority.

The applications may also be executed using the following command:

```
"java -Djava.security.manager -Djava.security.policy=policy -classpath . "package".class"
```

The package names and main classes for each component are specified in Table 5.1.

Component	Package	Main Class
Election Authority	<i>ea</i>	<i>GUI</i>
Bulletin Board	<i>bb</i>	<i>BulletinBoardServer</i>
Decryption Server	<i>ds</i>	<i>DSGUI</i>
Voter Application	<i>voter</i>	<i>GUI</i>

Table 5.1: Package names and main classes for the components in the voting system.

If any problems occur using the voting system in Linux, this may be related to how Java RMI uses the *hosts* file found in */etc/hosts*. The *hosts* file may need to be altered so that it links *localhost* to the external IP-address of the computer, instead of the loopback-address.

## 5.10 Summary

This chapter explained the implementation of the voting system in Java. All the components of the system were thoroughly explained with class diagrams and sequence charts. The next chapter will analyze the performance of the system and evaluate the feasibility of using this system for elections at NTNU.



## Chapter 6

# Performance Evaluations

In this chapter the performance of the voting system is tested, using different parameters and election types. The tests done in this chapter were performed on a 3 GHz Pentium D with 2 Gb RAM.

### 6.1 Comparison of Voting Methods

#### 6.1.1 Parameters for the Performance Evaluation

The parameters of these tests are chosen to suit the needs of an election at NTNU. Choosing the security parameters of a cryptosystem will always have to be a trade-off between performance and security. The parameters used in these tests would suffice if the election were to be held today, but as the computing power of computers increases with time, these parameters will have to be changed accordingly. For elections with higher security demands, the key length will have to be increased. A few tests with higher security demands are shown in Section 6.2.

If a larger  $M$  is needed, this may imply an increase in the block size needed to avoid overflow when calculating the results (remember that  $M^L < n^s$  must be satisfied when using the normal encoding of votes, and  $M^{2^{l+1}} < n^s$  when using the binary encoding of votes). The straight forward way of increasing the block size (the way that is performed automatically on-demand in the implementation of the system) is by increasing the parameter  $s$ . This implies doubling the block size of the cryptosystem, resulting in an increased complexity of the calculations. If only a small extension of the message space is needed, it may be more efficient to increase  $n$ .

$k$ (bitlength of $n$ )	1024
$s$	1
$q$ (security parameter)	80
$M$	10000

Table 6.1: The parameters chosen for the performance testing.

For most elections at NTNU, the values chosen in Table 6.1 should suffice. In most cases the value  $M$  can be increased without needing to increase  $s$ . For finding the  $s$  needed for a particular choice of parameters, the numbers can be inserted into the equation  $M^L < n^s$  ( $M^{2^{l+1}} < n^s$  when using the binary encoding of votes) and choosing  $s$  as the lowest value where this equation is true.

### 6.1.2 Performance Evaluation of Share Decryption and Vote Validation

The vote validation process is the bottleneck of the system. The decryption server needs to validate every vote cast, and then multiply the ciphertexts. As we want the election to be universally verifiable for anyone, a performance evaluation of this step is of utmost importance for choosing the right method for a given set of parameters.

This thesis has described two distinct ways of carrying out 1-out-of- $L$  elections. The two variants can perform the same elections, but have completely different ways of encoding votes and creating the zero-knowledge proofs of correctness. As explained in Section 4.5 the normal way of encoding votes requires  $O(L)$  proofs while the binary encoding only requires  $O(\log_2 L)$  proofs.

The binary encoding variant requires  $\log_2 L$  multiplication-mod- $n^s$  proofs and  $\log_2 L$  1-out-of-2  $n^s$ 'th proofs to prove correctness of a vote. Each multiplication-mod- $n^s$  proof requires two encryptions and two calculations of large numbers. Each 1-out-of-2  $n^s$ 'th proof also requires two encryptions and two calculations for verification.

The normal way of encoding needs only one 1-out-of- $L$   $n^s$ 'th proof for each vote. The verification of a 1-out-of- $L$   $n^s$ 'th requires  $L$  encryptions and  $L$  calculations of large integers. These calculations are less complex than those of the multiplication-mod- $n^s$  proofs used in the binary encoding of votes, and this leads to a slightly quicker verification of each proof.

Since the calculations done in the normal encoding are a small fraction quicker, these two methods are close to being equivalent for small  $L$ . As explained above, the binary encoding variant has a smaller complexity as  $L$  grows larger. Thus, it is first with a larger  $L$  that the binary encoding variant really starts to outshine the normal encoding variant. This leads to a large performance gain for large  $L$ , as shown in Table 6.2.

$L$	1-out-of- $L$ normal	1-out-of- $L$ binary
2	381 ms.	380 ms.
8	1226 ms.	1745 ms.
16	2330 ms.	2433 ms.
32	4580 ms.	3109 ms.
64	8933 ms.	3772 ms.
128	74264 ms.	4471 ms.

Table 6.2: The time spent for validation of votes and vote multiplication per vote for different values of  $L$ . The value is calculated as: Mean time spent on vote validation and vote multiplication for all decryption servers divided by the total number of votes given. This result can be calculated using the simulation tool, and is shown as: *Mean share decryption time per vote*.

This comparison is valid for choosing the right voting method in the case of a  $t$ -out-of- $L$  election as well. The results of simulations of 2-out-of- $L$  elections with varying  $L$  using the two different encodings can be seen in Table 6.3.

$L$	2-out-of- $L$ normal	2-out-of- $L$ binary
8	2831 ms.	3881 ms.
16	5082 ms.	5244 ms.
32	9534 ms.	6624 ms.
64	18270 ms.	7972 ms.

Table 6.3: A comparison of mean share decryption time per vote in 2-out-of- $L$  elections using the two different encodings.

It is clear that the same tendency can be seen in the case of a multi-vote election. The normal encoding of votes perform slightly better for small values of  $L$ , and the binary encoding of votes performs best for large values of  $L$ . This result was expected since the multi-vote election is realized as a parallel run of single vote elections, including cross validation of proofs.

### 6.1.3 Performance Evaluation of Vote Creation

The vote creation process includes the encryption of the votes and the creation of the zero-knowledge proofs. The computations done during this process should be

done as efficiently as possible because voters should be able to cast votes, without specific requirements to the computers used. A comparison of vote creation time using the two different encoding methods for votes is shown in Table 6.4 and Table 6.5.

$L$	1-out-of- $L$ normal	1-out-of- $L$ binary
2	414 ms.	1336 ms.
8	1321 ms.	2257 ms.
16	2603 ms.	3209 ms.
32	5525 ms.	4131 ms.
64	12448 ms.	5026 ms.
128	104059 ms.	5979 ms.

Table 6.4: A comparison of mean vote encryption and proof creation time in 1-out-of- $L$  elections using the two different encodings.

$L$	2-out-of- $L$ normal	2-out-of- $L$ binary
8	3396 ms.	5285 ms.
16	6026 ms.	7129 ms.
32	11864 ms.	9003 ms.
64	25707 ms.	10806 ms.

Table 6.5: A comparison of the two different encodings with respect to mean vote encryption and proof creation time in 2-out-of- $L$  elections.

It is clear that the same tendency is seen here as in the simulations in Section 6.1.2. Vote encryption and proof creation is quickest using the normal encoding if  $L$  is small. When  $L$  grows the binary encoding variant is quicker, because of the lower number of proofs needed.

#### 6.1.4 Performance of the Election Authority

The main functions of the election authority in this system is the key generation and combining valid shares acquired from decryption servers to a result. As explained in Section 5.2 the keys are generated by choosing random prime numbers and checking if they meet the requirements. This may be a time-consuming process, as the methods used for checking if a number is a prime requires much computing power when the bitlength of  $n$  is large. However, this should not be a major concern, since the key generation only needs be performed once for each election.

The election authority verifies the zero-knowledge proofs proving that the exponentiation done in the share decryption process is done correctly. The election

authority validates all votes cast, and checks that the decryption servers have calculated their share from the correct set of valid votes. Finally, the zero-knowledge proofs for the exponentiations of the result are verified. This simulation results for this process for different values of  $L$  and different voting methods are included in Table 6.6.

$L$	1-out-of- $L$ normal	1-out-of- $L$ binary	2-out-of- $L$ normal	2-out-of- $L$ binary
2	6718 ms.	6812 ms.	Not Available	Not Available
8	15078 ms.	20484 ms.	29375 ms.	39985 ms.
16	26266 ms.	27406 ms.	51797 ms.	53656 ms.
32	49016 ms.	34188 ms.	96032 ms.	67344 ms.
64	92829 ms.	40906 ms.	182954 ms.	80909 ms.
128	765666 ms.	47703 ms.	Not Available	Not Available

Table 6.6: A comparison of share combination time in all four voting methods.

The values in Table 6.6 shows a clear correlation to the tables in Section 6.1.3 and 6.1.2. The reason for this is the time spent validating votes. As we can see, the performances of the two voting methods is similar for  $L = 16$ , but the normal encoding is slightly quicker for  $L < 16$ , and the binary encoding is quicker for  $L > 16$ .

## 6.2 Elections with Higher Security Requirements

For evaluating the possibility of running elections with higher security demands, the system was tested out using a keylength of 2048 bits. This clearly leads to a degradation of performance because of the more complex computations. The remaining parameters were chosen as explained in Table 6.1. Table 6.7 shows the mean time needed to validate one vote using the  $L$  shown in the table using the two different encodings of votes in a 1-out-of- $L$  election.

$L$	1-out-of- $L$ normal	1-out-of- $L$ binary
2	2842 ms.	7849 ms.
8	9076 ms.	12848 ms.
16	17405 ms.	17859 ms.
32	33020 ms.	22931 ms.

Table 6.7: A comparison of mean share decryption and vote validation times per vote using a 2048-bits key.

As shown in this table, the vote validation times are much higher when using a 2048 bits key, but for elections with smaller  $L$  this should be computationally feasible.

### 6.3 Feasibility Analysis

A fully functional voting system has been implemented, and this section will discuss the feasibility of using this system in an election at NTNU.

A university performs a variety of elections for electing persons for different positions at the university, or internally at different faculties or institutes. The largest elections, such as the rectorial election may allow all students and employees to vote. NTNU has around 3500 employees and 20000 students, which would be the maximum number of voters in an election. We will now assume that everyone casts a vote. This will result in a total count of 23500 votes.

We give some examples of the vote validation time in different scenarios given the parameters mentioned in Section 6.1.1, but with  $M = 23500$ . All these examples satisfy  $M^L < n$  ( $M^{2^{L+1}} < n$  for the binary encoding of votes), which allows us to use  $s = 1$

A Yes/No-election using the normal encoding of votes would take around 2.5 hours to validate.

An 1-out-of- $L$  election with  $L = 8$  and normal encoding of votes, the validation of all votes would take around 8 hours.

In an 1-out-of- $L$  election with binary encoding of votes and  $L = 32$ , the total validation process would take approximately 20 hours.

In a 2-out-of- $L$  election using ranked voting with normal encoding of votes and  $L = 8$ , the validation of all votes would take between 18 and 19 hours.

In an example with a 2-out-of- $L$  election, binary encoding of votes and  $L = 32$  the validation of all votes would take around 43 hours.

These examples show that even elections with complex parameters and a large  $M$  can be validated in a reasonable amount of time using a normal desktop computer. It should be mentioned that the election turnout of former elections at NTNU have been much lower than 100 percent. This value was chosen for performance evaluation of the system in a "worst case scenario" where all the voters cast their votes. In practice, the election turnout would be much lower, which would result in a shorter time needed for the validation of all votes.

Choosing the right encoding is important for attaining the highest level of performance possible for a given set of parameters. For  $L < 16$  the normal encoding performs best because of the slightly less demanding calculations needed for each proof. For  $L > 16$  the lower number of proofs needed for the binary encoding is noticed, and this given a better performance than using the normal encoding.

## 6.4 Programming Language Comparison

All the tests were done using an implementation of the complete voting system in Java. For evaluation of the performance of Java as programming language when doing calculations with large integers, some tests were done with execution of the same operations using both Java and C. The library GNU MP<sup>1</sup> was used for performing the computations in the C version. The two programs were programmed to execute unoptimized encryptions of a given message using a 1024 bits modulo.

The results showed that the C version was consistently 3-4 times faster than the Java variant of the program. This leads us to think that the results of the time for validation of results could be reduced drastically by implementing the system in C or C++ instead of Java. An implementation of the system in C or C++ would not run as easily as the Java variant in a heterogeneous environment, but would have much better performance. This would allow more demanding election types to be carried out, with the elections still being considered universally verifiable.

## 6.5 Summary

This chapter provided a performance analysis of the voting system using realistic parameters that may be used for elections at NTNU. The two different encodings, normal and binary, performed differently depending on the number of candidates  $L$ . Because the calculations done using binary encoding are slightly more complex, the performance of normal encoding is better for small  $L$ . As  $L$  increases the number of proofs needed for verification of an election using normal encoding increases linearly, while the proofs needed using binary encoding increases logarithmically. The two encodings perform almost identically for  $L = 16$ . For  $L > 16$  the binary encoding performs better than the normal encoding.

---

<sup>1</sup>GNU multiple precision arithmetic library [24]

A comparison between C and Java in terms of performance in large integer calculations was done, and the C program performed consistently 3-4 times faster than the Java version. This means that a C or C++ version of the voting system could provide verification of votes much faster, and could allow more complex elections.

## Chapter 7

# Concluding Remarks

### 7.1 Conclusion

This thesis has described, designed, implemented and tested a voting system based on homomorphic encryption. The voting system is built on top of a generalization of Paillier's public-key system, and uses zero-knowledge proofs for proving correctness of votes, without revealing anything about the contents of the votes. The system has been implemented in Java, and the computational requirements for carrying out an election has been tested with various parameters which may be used in NTNU elections.

The design of the voting system maintains the privacy of the votes, is secured from cheating (by people both inside and outside of the electoral apparatus), is universally verifiable for every participant in the election and works well on all machines capable of running Java.

The system may be used to carry out Yes/No-elections, elections where a vote is cast for one out of  $L$  candidates and elections where  $t$  votes are cast for  $L$  candidates. The latter may be used to provide ranked elections, where votes are differently weighted when results are calculated. The voting methods were designed using two different encodings of votes, which had different performance on different sets of election parameters.

The most time-consuming task in an election is the validation of votes. For an election to be universally verifiable, the validation of votes should be computationally feasible for anyone, without the need of special hardware to speed up the compu-

tations. The main reason for making an election verifiable is not necessarily that everyone should do it, but that everyone should *be able* to do it. Using a 3 GHz Pentium D with 2 Gb RAM and a 1024 bit key, even a 2-out-of- $L$  election with  $L = 32$  with 23500 participants was verifiable in less than two days, when testing the system presented in this thesis.

A few tests comparing the performance of Java and C as programming languages for large integer computations, were undertaken. The C version was consistently 3-4 times faster than the Java version, implying that higher performance may be achieved by implementing the voting system in C or C++.

## 7.2 Future Work

Designing, implementing and testing several types of voting systems is important in order to evaluate the security and performance of different voting schemes. Research should be undertaken in order to compare the different schemes and to evaluate how well they function in different types of scenarios.

The binary encoding of votes used in this thesis requires the chosen parameters to satisfy  $M^{2^{l+1}} < n^s$ . There are two drawbacks associated with this: it allows voters to cast votes for non-existing candidates, and it may also, in some scenarios, require the use of a block size which is twice as large as what would otherwise have been needed. Adding an extra step in the verification of a vote, where the voter proves in zero-knowledge that the chosen candidate is valid, may inhibit both of the drawbacks mentioned. The procedure for doing this is described in [19]. A performance evaluation of the binary encoding of votes, using this technique, may be interesting.

The implementation of the system in Java does not seem to be as efficient as an implementation in C or C++. The core functions of the system may easily be translated to C or C++. Doing this will probably imply a large performance gain, and may allow more complex elections to be universally verifiable in a shorter amount of time.

The electronic voting system presented in this thesis is probably more secure than most existing electronic voting systems used in elections at universities and in organizations today. The implementation in this thesis may be integrated at NTNU today, and tested in a real election. Certain elements that were outside the scope

of this thesis, like authentication of voters and access control of the system, would have to be implemented in order to make this voting system to function securely. It would be interesting to see a universally verifiable voting system, like the one presented in this thesis, be put into use in a real election outside the lab.



## Bibliography

- [1] Alessandro Acquisti. Receipt-free homomorphic elections and write-in ballots. Cryptology ePrint Archive, Report 2004/105, 2004. <http://eprint.iacr.org/>.
- [2] Riza Aditya, Byoungcheon Lee, Colin Boyd, and Ed Dawson. An efficient mixnet-based voting scheme providing receipt-freeness. In Sokratis K. Katsikas, Javier Lopez, and Günther Pernul, editors, *TrustBus*, volume 3184 of *Lecture Notes in Computer Science*, pages 152–161. Springer, 2004.
- [3] Mihir Bellare and Phillip Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In *ACM Conference on Computer and Communications Security*, pages 62–73, 1993.
- [4] D. Bernstein. Circuits for integer factorization: a proposal, 2001. [cr.yp.to/papers.html#nfsccircuit](http://cr.yp.to/papers.html#nfsccircuit), Visited June 5. 2006.
- [5] CyberVote. The history of electronic voting. [http://www.eucybervote.org/Reports/KUL-WP2-D4V1-v1.0-01.htm#P323\\_14632](http://www.eucybervote.org/Reports/KUL-WP2-D4V1-v1.0-01.htm#P323_14632), Visited February 20. 2006.
- [6] Ivan Damgård, Jens Groth, and Gorm Salomonsen. The theory and implementation of an electronic voting system. In D. Gritzalis, editor, *Secure Electronic Voting*, pages 77–100. Kluwer Academic Publishers, 2003.
- [7] Donald E. Eastlake, Stephan D. Crocker, and Jeffrey I. Schiller. RFC1750: Randomness requirements for security. Technical Report 1750, 1994.
- [8] Amos Fiat and Adi Shamir. How to prove yourself: Practical solutions to identification and signature problems. In Andrew M. Odlyzko, editor, *CRYPTO*, volume 263 of *Lecture Notes in Computer Science*, pages 186–194. Springer, 1986.
- [9] Jens Groth. Honest verifier zero-knowledge arguments applied. ds DS-04-3, BRICS - Basic Research in Computer Science, October 2004. PhD thesis. xii+119 pp.
- [10] Mads J. Jurik. Progress report, May 2001. <http://www.brics.dk/~jurik/Research/progress.ps>, Visited June 15. 2006.

- [11] Sun Microsystems. Java Remote Method Invocation (Java RMI). <http://java.sun.com/products/jdk/rmi/>, Visited May 10. 2006.
- [12] Sun Microsystems. Java Secure Socket Extension (JSSE). <http://java.sun.com/products/jsse/>, Visited May 10. 2006.
- [13] D. Naccache. Double-speed safe prime generation. Crypto Eprint Archive, entry 2003:175, <http://eprint.iacr.org>, 2003.
- [14] NIST. Key management guideline - workshop document. draft, 2001. [http://csrc.nist.gov/encryption/kms/key-management-guideline-\(workshop\).pdf](http://csrc.nist.gov/encryption/kms/key-management-guideline-(workshop).pdf).
- [15] NTNU. Innsida. <https://innsida.ntnu.no/>, Visited June 7, 2006.
- [16] Tatsuaki Okamoto. Receipt-free electronic voting schemes for large scale elections. In Bruce Christianson, Bruno Crispo, T. Mark A. Lomas, and Michael Roe, editors, *Security Protocols Workshop*, volume 1361 of *Lecture Notes in Computer Science*, pages 25–35. Springer, 1997.
- [17] The Open SSL Project. OpenSSL Homepage. <http://www.openssl.org/>, Visited June 7, 2006.
- [18] Pascal Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *EUROCRYPT*, pages 223–238, 1999.
- [19] Ivan Damgård, Mads Jurik, and Jesper Buus Nielsen. A generalization of Paillier’s public-key system with applications to electronic voting. Technical report, BRICS - Basic Research in Computer Science, 2003.
- [20] Berry Schoenmakers. A simple publicly verifiable secret sharing scheme and its application to electronic voting. In Michael J. Wiener, editor, *CRYPTO*, volume 1666 of *Lecture Notes in Computer Science*, pages 148–164. Springer, 1999.
- [21] RSA Security. RSA Security homepage. <http://rsasecurity.com>, Visited June 9. 2006.
- [22] Michael Ian Shamos. Electronic voting - evaluating the threat. In *CFP’93*, 1993.
- [23] Victor Shoup. Practical threshold signatures. In *EUROCRYPT*, pages 207–220, 2000.

- [24] Free Software Foundation. GNU MP Homepage. <http://swox.com/gmp/>,  
Visited May 10, 2006.



# Appendix A

## Source Code

### A.1 Voter Application

This section describes the methods used for encryption of votes and creation of the non-interactive zero-knowledge proofs of knowledge needed to prove correctness of votes. Here, only the most important methods used in the voting system are described. The full source code is included in the digital appendix following this thesis, described in Appendix B.

#### A.1.1 Precomputation of Values for Encryption

```
2  /** Method for precomputing values used in the optimized encryption algorithm  
3  *  
4  * @return  
5  */  
6  private BigInteger [] preCompute() {  
7      BigInteger precomp[] = new BigInteger[s + 1];  
8      BigInteger jFac = BigInteger.ONE;  
9      precomp[1] = n;  
10     for (int j = 1; j <= s; j++) {  
11         jFac = jFac.multiply(BigInteger.valueOf(j));  
12         precomp[j] = jFac.modInverse(nArray[s + 1]);  
13         precomp[j] = precomp[j].multiply(n.pow(j)).mod(nArray[s + 1]);  
14     }  
15     return precomp;  
16 }
```

Listing A.1: Precomputation of values for encryption

### A.1.2 Optimized Encryption Algorithm

```

2  /** The optimized encryption algorithm
3  *
4  * @param candidate
5  * @return The ciphertext and the random value used to hide the plaintext
6  */
7  public BigInteger[] encryptOptimized(int candidate) {
8      BigInteger[] encryption = new BigInteger [2];
9      BigInteger voteValue = voteValues[ candidate ];
10     BigInteger c = BigInteger.ONE.add(voteValue.multiply(n));
11     BigInteger temp = voteValue;
12     for (int j = 2; j <= s; j++) {
13         BigInteger j_ = BigInteger.valueOf(j);
14         temp = temp.multiply (( voteValue . subtract ( j_ )). add( BigInteger .ONE))
15             .mod(nArray[s - j + 1]);
16         c = (c.add(temp.multiply (precomp[j ]))). mod(nArray[s + 1]);
17     }
18     BigInteger v = new BigInteger(n.bitCount (), rand);
19     while (v.compareTo(n) >= 0 || v.gcd(n).compareTo(BigInteger.ONE) > 0)
20         v = new BigInteger(n.bitCount (), rand);
21
22     BigInteger rns = v.modPow(nArray[s], nArray[s + 1]);
23     encryption [0] = c . multiply ( rns ) . mod(nArray[s + 1]);
24     encryption [1] = v;
25
26     return encryption ;
27 }

```

Listing A.2: Full optimized encryption algorithm

### A.1.3 Creation of a Vote in a 1-out-of-L Election

```

2  /**
3  * The method for encrypting a Vote and creating the proof of correctness
4  * in the case of 1-out-of-L elections
5  *
6  * @param candidate
7  * The candidate the Client voted for
8  * @return The Vote including the proof of correctness
9  * @throws IncorrectVoteException
10 */
11 public Vote encryptVote(int candidate) throws IncorrectVoteException {
12     if ( candidate > L)
13         throw new IncorrectVoteException("Choice out of bounds");
14
15     BigInteger g = pk.getG();
16     BigInteger z[] = new BigInteger[L];
17     BigInteger a[] = new BigInteger[L];
18     BigInteger e[] = new BigInteger[L];

```

```

18 BigInteger valuesToBeHashed[] = new BigInteger[L + 2];
   valuesToBeHashed[0] = n;
20 valuesToBeHashed[1] = BigInteger.valueOf(id);

22 BigInteger r = null;
   BigInteger e_sum = BigInteger.ZERO;
24 BigInteger pow_t = BigInteger.valueOf(2).pow(t);

26 BigInteger[] encryption = encryptOptimized(candidate);
   BigInteger ciphertext_vote = encryption[0];
28 voteRandom[votePointer] = encryption[1];
   BigInteger v = encryption[1];
30 for (int i = 0; i < L; i++) {
   if (i == candidate) {
32         // Invoke Protocol 1-out-of-k n's'th power for the real vote.

34         // Choose a random r in  $\mathbb{Z}_n^{s+1}$  to use in  $a[i] = E(0, r)$ 
         r = new BigInteger(n.bitCount(), rand);
36         while (r.compareTo(n) >= 0
               || r.gcd(n).compareTo(BigInteger.ONE) > 0)
         r = new BigInteger(n.bitCount(), rand);

40         // Calculate a[i]
         a[i] = r.modPow(nArray[s], nArray[s + 1]);

42     } else {
44         // Invoke the honest-verifier simulator for n's'th power
         // protocol

46         // Choose z[i] random in  $\mathbb{Z}_n$  rel.prime to n
         z[i] = new BigInteger(n.bitCount(), rand);
48         while (z[i].compareTo(n) >= 0
               || z[i].gcd(n).compareTo(BigInteger.ONE) > 0)
         z[i] = new BigInteger(n.bitCount(), rand);

52         // Choose e[i] random in  $\mathbb{Z}_{(2^t)}$ 
         e[i] = new BigInteger(t, rand);
54         e_sum = e_sum.add(e[i]).mod(pow_t);

56         // Set a[i] =
         // ( ciphertext_vote / g^votevalue[i] )^{(-e[i]) * z[i]^{(n^s)}} mod
         // n^{(s+1)}
60         BigInteger temp = (g.modPow(voteValues[i], nArray[s + 1]))
                           .modInverse(nArray[s + 1]);
62         a[i] = (ciphertext_vote.multiply(temp).modPow(e[i],
               nArray[s + 1])).modInverse(nArray[s + 1]);
64         a[i] = a[i].multiply(z[i].modPow(nArray[s], nArray[s + 1]));

66     }
   // Include all a[i]-values into the hash-function-values
68   valuesToBeHashed[i + 2] = a[i];

```

```

70 }
    // Generate a hash-value as challenge, using Fiat-Shamir heuristic with
72 // SHA-256
    BigInteger e_Challenge = h.generateHash(valuesToBeHashed, t);
74 // Calculate the random challenge for e[candidate]
    e[candidate] = (e_Challenge.subtract(e_sum)).mod(pow_t);
76 // Calculate z[candidate]
    z[candidate] = (r.multiply(v.modPow(e[candidate], nArray[s + 1])))
78     .mod(nArray[s + 1]);

80 return new Vote( ciphertext_vote , a, e_Challenge, z, e, id);
}

```

Listing A.3: Creation of a *Vote* in a 1-out-of-*L* Election

### A.1.4 Creation of a *MultipleVote* in a *t*-out-of-*L* Election

```

/** Encrypts the chosen votes using the method explained as k-out-of-L voting.
2 *
    * @param candidate chosen candidates
4 * @return The MultipleVotes object containing the votes and proofs needed.
    * @throws IncorrectVoteException If any choices were out of
6 *     bounds, or if incorrect number of votes were cast.
    */
8
    public MultipleVotes encryptMultipleVotes (int[] candidate)
10         throws IncorrectVoteException {

12     if (candidate.length != noOfVotes)
        throw new IncorrectVoteException("Incorrect number of votes cast");
14
        Vote[] votes = new Vote[candidate.length];
16
        for (; votePointer < candidate.length; votePointer++) {
18
            votes[votePointer] = encryptVote(candidate[votePointer]);
20
        }
22 // Public input: n, g, E(a), E(b), E(c) where a*b = c mod n
        BigInteger ea[][] = new BigInteger[candidate.length][candidate.length];
        BigInteger eb[][] = new BigInteger[candidate.length][candidate.length];
        BigInteger ec[][] = new BigInteger[candidate.length][candidate.length];
24 BigInteger ra[][] = new BigInteger[candidate.length][candidate.length];
        BigInteger rb[][] = new BigInteger[candidate.length][candidate.length];
        BigInteger rc[][] = new BigInteger[candidate.length][candidate.length];
        BigInteger A[][] = new BigInteger[candidate.length][candidate.length];
26 BigInteger B[][] = new BigInteger[candidate.length][candidate.length];

32 for (int i = 0; i < candidate.length; i++) {
    for (int j = i + 1; j < candidate.length; j++) {

```

```

34         A[i][j] = voteValues[candidate[i]]
                .subtract(voteValues[candidate[j]]);
36         B[i][j] = A[i][j].modInverse(nArray[s + 1]);
        BigInteger[] temp = encryptOptimized(A[i][j]);
38         ea[i][j] = (votes[i].getVote().multiply(votes[j].getVote()
                .modInverse(nArray[s + 1])).mod(nArray[s + 1])); // temp[0];
40         ra[i][j] = (voteRandom[i].multiply(voteRandom[j]
                .modInverse(nArray[s + 1])).mod(nArray[s + 1])); // temp[1];
42
        temp = encryptOptimized(B[i][j]);
44         eb[i][j] = temp[0];
        rb[i][j] = temp[1];
46
        temp = encryptOptimized(BigInteger.ONE);
48         ec[i][j] = temp[0];
        rc[i][j] = temp[1];
50
    }
52 }

54 BigInteger[][] d = new BigInteger[candidate.length][candidate.length];
    BigInteger[][] rd = new BigInteger[candidate.length][candidate.length];
56 BigInteger[][] ed = new BigInteger[candidate.length][candidate.length];
    BigInteger[][] edb = new BigInteger[candidate.length][candidate.length];
58 BigInteger[][] rdb = new BigInteger[candidate.length][candidate.length];
    BigInteger[][] e = new BigInteger[candidate.length][candidate.length];
60 BigInteger[][] f = new BigInteger[candidate.length][candidate.length];
    BigInteger[][] z1 = new BigInteger[candidate.length][candidate.length];
62 BigInteger[][] z2 = new BigInteger[candidate.length][candidate.length];

64 for (int i = 0; i < candidate.length; i++) {
    for (int j = i + 1; j < candidate.length; j++) {
66         d[i][j] = new BigInteger(nArray[s].bitLength(), rand);
        BigInteger[] temp2 = encryptOptimized(d[i][j]);
68         ed[i][j] = temp2[0];
        rd[i][j] = temp2[1];
70         temp2 = encryptOptimized(d[i][j].multiply(B[i][j]));
        edb[i][j] = temp2[0];
72         rdb[i][j] = temp2[1];

74         BigInteger[] valuesToBeHashed = new BigInteger[5];
        valuesToBeHashed[0] = n;
76         valuesToBeHashed[1] = ea[i][j];
        valuesToBeHashed[2] = eb[i][j];
78         valuesToBeHashed[3] = ec[i][j];
        valuesToBeHashed[4] = BigInteger.valueOf(id);
80         e[i][j] = h.generateHash(valuesToBeHashed, t);
        f[i][j] = (e[i][j].multiply(A[i][j])).add(d[i][j]).mod(
82             nArray[s]);

84         z1[i][j] = ((ra[i][j].modPow(e[i][j], n)).multiply(rd[i][j]))
                .mod(n);

```

```

86         z2[i][j] = rdb[i][j].multiply(rc[i][j].modPow(e[i][j], n));
            z2[i][j] = z2[i][j].modInverse(n);
88         z2[i][j] = (z2[i][j].multiply(rb[i][j].modPow(f[i][j],
            nArray[s + 1])).mod(n);
90     }
    }
92 MultipleVotes mVotes = new MultipleVotes(votes, ea, eb, ec, ed, edb, e,
    f, z1, z2, id);
94
    return mVotes;
96 }

```

Listing A.4: Creation of a *MultipleVote* in a *t*-out-of-*L* Election

### A.1.5 Creation of a *BinaryVote* in a 1-out-of-*L* Election

```

/** Encrypts a vote in the case of 1-out-of-L Binary Voting
2 *
    * @param candidate The candidate that was voted for
4 * @return The BinaryVote holding the ciphertext and the needed
    *         zero-knowledge proofs of correctness
6 * @throws IncorrectVoteException If the choice was out of bounds
    */
8 public BinaryVote encryptVoteBinary(int candidate)
    throws IncorrectVoteException {
10     if (candidate > L)
        throw new IncorrectVoteException("Choice out of bounds");
12
    l = 1;
14     BigInteger M_ = BigInteger.valueOf(M);
    while (Math.pow(2, l + 1) < L)
16         l++;

18     String bitRep = Integer.toString(candidate, 2);

20     BigInteger g = pk.getG();
    BigInteger pow_t = BigInteger.valueOf(2).pow(t);
22     BigInteger encryptions[] = new BigInteger[l + 1];
    BigInteger rb[] = new BigInteger[l + 1];
24     BigInteger [][] a = new BigInteger[2][l + 1];
    BigInteger [][] e = new BigInteger[2][l + 1];
26     BigInteger [][] z = new BigInteger[2][l + 1];

28     BigInteger valuesToBeHashed[] = new BigInteger[4];
    valuesToBeHashed[0] = n;
    valuesToBeHashed[1] = BigInteger.valueOf(id);
    BigInteger MraisedToBitValue[] = new BigInteger[l + 1];
32
    for (int i = 0; i <= l; i++) {
34         int biti = 0;

```

```

try {
36     biti = candidate % 2;
        candidate = candidate / 2;
38 } catch (Exception ex) {
        biti = 0;
40 }
    if (biti == 1) {
42         // The bit in question is 1. Create encryption and proofs for
            // this case.
44         // System.out.println ( biti );
        MraisedToBitValue[i] = M..pow((int) Math.pow(2, i ));
46         BigInteger[] tempVote = encryptOptimized(MraisedToBitValue[i ]);
        encryptions[i] = tempVote[0];
48         BigInteger v = tempVote[1];
        rb[i] = v;
50         // Choose a random r in  $Z_n^{(s+1)}$  to use in  $a[i] = E(0, r)$ 
        BigInteger r = new BigInteger(n.bitCount(), rand);
52         while (r.compareTo(n) >= 0
                || r.gcd(n).compareTo(BigInteger.ONE) > 0)
54             r = new BigInteger(n.bitCount(), rand);

56         // Calculate a[i]
        a[0][i] = r.modPow(nArray[s], nArray[s + 1]);
58
        // Create honest-verifier simulator for the other value.
        // Choose z[i] random in  $Z_n$  rel.prime to n
60         z[1][i] = new BigInteger(n.bitCount(), rand);
62         while (z[1][i].compareTo(n) >= 0
                || z[1][i].gcd(n).compareTo(BigInteger.ONE) > 0)
64             z[1][i] = new BigInteger(n.bitCount(), rand);

66         // Choose e[i] random in  $Z_{(2^t)}$ 
        e[1][i] = new BigInteger(t, rand);
68
        // Set a[i] =
70         // ( ciphertext_vote / g^votevalue[i] )^( -e[i] ) * z[i]^( n^s ) mod
            // n^(s+1)
72         BigInteger temp = g.modInverse(nArray[s + 1]);
        a[1][i] = ( encryptions[i].multiply(temp).modPow(e[1][i],
74                 nArray[s + 1])).modInverse(nArray[s + 1]);
        a[1][i] = a[1][i].multiply(z[1][i].modPow(nArray[s],
76                 nArray[s + 1]));

78         valuesToBeHashed[2] = a[0][i];
        valuesToBeHashed[3] = a[1][i];
80
        BigInteger e_Challenge = h.generateHash(valuesToBeHashed, t)
82             .mod(pow.t);

84         e[0][i] = e_Challenge.subtract(e[1][i]).mod(pow.t);
        ;
86         z[0][i] = (r.multiply(v.modPow(e[0][i], nArray[s + 1])))

```

```

88         .mod(nArray[s + 1]);
89     } else {
90         // The bit in question is 0. Create encryption and proofs for
91         // this case.
92         MraisedToBitValue[i] = BigInteger.ONE;
93         BigInteger [] tempVote = encryptOptimized( BigInteger.ONE);
94         encryptions [i] = tempVote[0];
95         BigInteger v = tempVote[1];
96         rb[i] = v;
97
98         // Choose a random r in Z_n to use in a[i] = E(0,r)
99         BigInteger r = new BigInteger(n.bitCount (), rand);
100        while (r.compareTo(n) >= 0
101            || r.gcd(n).compareTo(BigInteger.ONE) > 0)
102            r = new BigInteger(n.bitCount (), rand);
103
104        // Calculate a[i]
105        a[1][i] = r.modPow(nArray[s], nArray[s + 1]);
106
107        // Create honest-verifier simulator for the other value.
108        // Choose z[i] random in Z_n rel.prime to n
109        z[0][i] = new BigInteger(n.bitCount (), rand);
110        while (z[0][i].compareTo(n) >= 0
111            || z[0][i].gcd(n).compareTo(BigInteger.ONE) > 0)
112            z[0][i] = new BigInteger(n.bitCount (), rand);
113
114        // Choose e[i] random in Z_(2^t)
115        e[0][i] = new BigInteger(t, rand);
116
117        // Set a[i] = ( ciphertext_vote / g^(M*2^i))^( -e[i] ) * z[i]^(n^s) mod
118        // n^(s+1)
119        BigInteger temp = (calculateGM(M..pow((int) Math.pow(2, i))))
120            .modInverse(nArray[s + 1]);
121        a[0][i] = ( encryptions [i].multiply(temp).modPow(e[0][i],
122            nArray[s + 1])).modInverse(nArray[s + 1]);
123        a[0][i] = a[0][i].multiply(z[0][i].modPow(nArray[s],
124            nArray[s + 1]));
125
126        valuesToBeHashed[2] = a[0][i];
127        valuesToBeHashed[3] = a[1][i];
128
129        BigInteger e_Challenge = h.generateHash(valuesToBeHashed, t)
130            .mod(pow_t);
131        e[1][i] = e_Challenge.subtract(e[0][i]).mod(pow_t);
132        z[1][i] = (r.multiply(v.modPow(e[1][i], nArray[s + 1]))
133            .mod(nArray[s + 1]));
134    }
135 }
136
137 // Proofs for that bits are 1 or 0 are created. Now to the

```

```

// Proofs for the 2nd part
140 BigInteger Fi [] = new BigInteger[1 + 1];
    BigInteger fi [] = new BigInteger[1 + 1];
142 BigInteger rfi [] = new BigInteger[1 + 1];

144 Fi[0] = MraisedToBitValue[0];
    fi[0] = encryptions[0];
146 rfi[0] = rb[0];

148 BigInteger [] tempVote;
    for (int i = 1; i <= 1; i++) {
150         Fi[i] = Fi[i - 1].multiply(MraisedToBitValue[i]);
            tempVote = encryptOptimized(Fi[i]);
152         fi[i] = tempVote[0];
            rfi[i] = tempVote[1];
154         if (i == 1) {
            voteRandom[votePointer] = rfi[i];
156         }
    }
158

// Make the multiplication proof!
160 BigInteger z1 [] = new BigInteger[1];
    BigInteger z2 [] = new BigInteger[1];
162 BigInteger d [] = new BigInteger[1];
    BigInteger ed [] = new BigInteger[1];
164 BigInteger edb [] = new BigInteger[1];
    BigInteger rdb [] = new BigInteger[1];
166 BigInteger rd [] = new BigInteger[1];
    BigInteger eMult [] = new BigInteger[1];
168 BigInteger f [] = new BigInteger[1];
    BigInteger [] valuesToBeHashed2 = new BigInteger[5];
170 valuesToBeHashed2[0] = n;
    valuesToBeHashed2[4] = BigInteger.valueOf(id);
172 for (int i = 0; i < 1; i++) {
        d[i] = new BigInteger(nArray[s].bitLength(), rand);
        BigInteger [] temp2 = encryptOptimized(d[i]);
174         ed[i] = temp2[0];
            rd[i] = temp2[1];
            temp2 = encryptOptimized(d[i].multiply(encryptions[i + 1]));
176         edb[i] = temp2[0];
            rdb[i] = temp2[1];
178
180         valuesToBeHashed2[1] = fi[i];
            valuesToBeHashed2[2] = encryptions[i + 1];
            valuesToBeHashed2[3] = fi[i + 1];
182
184         eMult[i] = h.generateHash(valuesToBeHashed2, t);
186
188         f[i] = (eMult[i].multiply(Fi[i])).add(d[i]).mod(nArray[s]);
            z1[i] = ((rfi[i].modPow(eMult[i], n)).multiply(rd[i])).mod(n);
            z2[i] = rdb[i].multiply(rfi[i + 1]).modPow(eMult[i], n);
190         z2[i] = z2[i].modInverse(nArray[s + 1]);

```

```

192         z2[i] = (z2[i].multiply(rb[i + 1].modPow(f[i], nArray[s + 1])))
               .mod(n);
194     }

196     return new BinaryVote(encryptions, e, z, a, fi, z1, z2, eMult, ed, edb,
                           f, id);
198 }

```

Listing A.5: Creation of a *BinaryVote* in a 1-out-of- $L$  election

### A.1.6 Creation of a *MultipleBinaryVote* in a $t$ -out-of- $L$ Election

```

/** Encrypts the chosen votes using the method explained as k-out-of-L Binary encoded voting.
2  *
   * @param candidate chosen candidates
4  * @return The MultipleBinaryVotes object containing the votes and proofs needed.
   * @throws IncorrectVoteException If any choices were out of bounds, or if
6  *      incorrect number of votes were cast.
   */
8  public MultipleBinaryVotes encryptMultipleVoteBinary (int candidate [])
   throws IncorrectVoteException {
10     BigInteger M_ = BigInteger.valueOf(M);
   if (candidate.length != noOfVotes)
12         throw new IncorrectVoteException("Incorrect number of votes cast");
   for (int i = 0; i < candidate.length; i++)
14         System.out.println (candidate[i]);
   BinaryVote[] votes = new BinaryVote[candidate.length];
16
   for (votePointer = 0; votePointer < candidate.length; votePointer++) {
18         votes[votePointer] = encryptVoteBinary (candidate[votePointer]);
   }
20     // Public input: n, g, E(a), E(b), E(c) where a*b = c mod n
   BigInteger ea [][] = new BigInteger[candidate.length][candidate.length];
22   BigInteger eb [][] = new BigInteger[candidate.length][candidate.length];
   BigInteger ec [][] = new BigInteger[candidate.length][candidate.length];
24   BigInteger ra [][] = new BigInteger[candidate.length][candidate.length];
   BigInteger rb [][] = new BigInteger[candidate.length][candidate.length];
26   BigInteger rc [][] = new BigInteger[candidate.length][candidate.length];
   BigInteger A [][] = new BigInteger[candidate.length][candidate.length];
28   BigInteger B [][] = new BigInteger[candidate.length][candidate.length];

30   for (int i = 0; i < candidate.length; i++) {
       for (int j = i + 1; j < candidate.length; j++) {
32
           A[i][j] = ((M_.modPow(BigInteger.valueOf(candidate[i]),
34                     nArray[s + 1])).subtract (M_.modPow(BigInteger
                       .valueOf(candidate[j]), nArray[s + 1])));
36

```

```

38         B[i][j] = A[i][j].modInverse(nArray[s + 1]).mod(nArray[s + 1]);
        BigInteger[] temp = encryptOptimized(A[i][j]);

40
        BigInteger temp2 = votes[j].getFi(0)[1]
42            .modInverse(nArray[s + 1]);
        ea[i][j] = ((votes[i].getFi(0)[1]).multiply(temp2))
44            .mod(nArray[s + 1]);
        ra[i][j] = (voteRandom[i].multiply(voteRandom[j]
46            .modInverse(nArray[s + 1]))).mod(nArray[s + 1]); // temp[1];

        temp = encryptOptimized(B[i][j]);
        eb[i][j] = temp[0];
50        rb[i][j] = temp[1];

52        temp = encryptOptimized(BigInteger.ONE);
        ec[i][j] = temp[0];
54        rc[i][j] = temp[1];

56    }
    }

58    BigInteger[][] d = new BigInteger[candidate.length][candidate.length];
    BigInteger[][] rd = new BigInteger[candidate.length][candidate.length];
    BigInteger[][] ed = new BigInteger[candidate.length][candidate.length];
62    BigInteger[][] edb = new BigInteger[candidate.length][candidate.length];
    BigInteger[][] rdb = new BigInteger[candidate.length][candidate.length];
64    BigInteger[][] e = new BigInteger[candidate.length][candidate.length];
    BigInteger[][] f = new BigInteger[candidate.length][candidate.length];
66    BigInteger[][] z1 = new BigInteger[candidate.length][candidate.length];
    BigInteger[][] z2 = new BigInteger[candidate.length][candidate.length];
68
    for (int i = 0; i < candidate.length; i++) {
        for (int j = i + 1; j < candidate.length; j++) {
70            d[i][j] = new BigInteger(nArray[s].bitLength(), rand);
            BigInteger[] temp2 = encryptOptimized(d[i][j]);
72            ed[i][j] = temp2[0];
            rd[i][j] = temp2[1];
74            temp2 = encryptOptimized(d[i][j].multiply(B[i][j]));
            edb[i][j] = temp2[0];
76            rdb[i][j] = temp2[1];

78            BigInteger[] valuesToBeHashed = new BigInteger[5];
            valuesToBeHashed[0] = n;
            valuesToBeHashed[1] = ea[i][j];
82            valuesToBeHashed[2] = eb[i][j];
            valuesToBeHashed[3] = ec[i][j];
            valuesToBeHashed[4] = BigInteger.valueOf(id);
84            e[i][j] = h.generateHash(valuesToBeHashed, t);

86            f[i][j] = (e[i][j].multiply(A[i][j])).add(d[i][j]).mod(
88                nArray[s]);

```

```

90         z1[i][j] = ((ra[i][j].modPow(e[i][j], n)).multiply(rd[i][j]))
                    .mod(n);
92         z2[i][j] = rdb[i][j].multiply(rc[i][j].modPow(e[i][j], n));
        z2[i][j] = z2[i][j].modInverse(n);
94         z2[i][j] = (z2[i][j].multiply(rb[i][j].modPow(f[i][j],
                    nArray[s + 1]))).mod(n);
96     }
    }
98 MultipleBinaryVotes mVotes = new MultipleBinaryVotes(votes, ea, eb, ec,
        ed, edb, e, f, z1, z2, id);
100
    return mVotes;
102
}
```

Listing A.6: Creation of a *MultipleBinaryVote* in a *t*-out-of-*L* election

## A.2 The Share Decryption of Votes

This method is executed by the decryption server to calculate the decrypted share of the result. The calculations use the information stored in the *KeyShare* object assigned to this specific decryption server.

```

/**
2 * The method for decrypting the results of the election with the secret
  * KeyShare
4 *
  * @param votes
6 *     The votes from the election
  */
8 public DecryptedShare shareDecrypt(Vote[] votes)
    throws NoValidVotesException {
10     int bitlength = (s + 2) * k + t;
    ArrayList validVotes = new ArrayList();
12
    for (int i = 0; i < votes.length; i++) {
14         if (val.checkProof(votes[i]))
            validVotes.add(votes[i]);
16     }

18     if (validVotes.size() == 0)
        throw new NoValidVotesException("No valid votes given");
20     // Calculate encrypted result of the election
    BigInteger result = ((Vote) validVotes.get(0)).getVote();
22     for (int i = 1; i < validVotes.size(); i++) {
        result = result.multiply(((Vote) validVotes.get(i)).getVote()).mod(
24         nArray[s + 1]);
    }
}
```

```

26     BigInteger c = result ;
    BigInteger ci = c.modPow(secretExponent, nArray[s + 1]);
28
    BigInteger random = new BigInteger( bitlength , r );
30    // Generate zero-knowledge proof
    BigInteger a = (c.pow(4)).mod(nArray[s + 1]);
32    a = a.modPow(random, nArray[s + 1]);

34    BigInteger b = v.modPow(random, nArray[s + 1]);

36    BigInteger e;
    BigInteger [] values = new BigInteger [5];
38    values [0] = n;
    values [1] = a;
40    values [2] = b;
    values [3] = c.pow(4).mod(nArray[s + 1]);
42    values [4] = ci.pow(2).mod(nArray[s + 1]);

44    e = h.generateHash( values , t );
    BigInteger temp2 = (e.multiply ( si ).multiply ( delta ));
46    BigInteger z = random.add(temp2);

48    DecryptedShare ds = new DecryptedShare(e, z, c, ci, id);
    return ds;
50 }

```

Listing A.7: Share decryption of votes and creation of a zero-knowledge proof for the calculations

### A.3 The Verification of the Zero-Knowledge Proofs

The methods included in this section is located in the *Validator* class in the *votingSystem* package of the system. These methods are used for validation of votes and decrypted shares.

#### A.3.1 Verification of a *Vote* in a 1-out-of-*L* Election

```

/**
2 * The code for checking if a Vote is of the correct form.
 *
4 * @param v The vote that will be checked
 *
6 * @return true if the proof is correct.
 */
8 public boolean checkProof(Vote v) {

```

```

10     BigInteger [] e = v.getE ();
    BigInteger [] z = v.getZ ();
12     BigInteger [] a = v.getA ();
    BigInteger e_challenge = v.getChallenge ();
14     // The variable to hold the sum of e_i values
    BigInteger e_sum = BigInteger.ZERO;

16
    // The array that holds the values to be used in the hash-function
18     BigInteger valuesToBeHashed[] = new BigInteger[L + 2];
    valuesToBeHashed[0] = n;
20     valuesToBeHashed[1] = BigInteger.valueOf(v.getId ());

22     BigInteger voteCheckValue1, voteCheckValue2;

24
    // Check all e,a,z values to see if they are correct.
26     for (int i = 0; i < L; i++) {
        e_sum = e_sum.add(e[i ]). mod(pow.t);

28
        voteCheckValue1 = normalVoteValues[i ]. multiply (v.getVote ());
30        voteCheckValue1 = a[i ]. multiply (
            voteCheckValue1.modPow(e[i], nArray[s + 1])). mod(
32            nArray[s + 1]);

34        voteCheckValue2 = z[i ]. modPow(nArray[s], nArray[s + 1]);

36        if (voteCheckValue1.compareTo(voteCheckValue2) != 0)
            return false ;

38
        if ((n.gcd(a[i ])). compareTo(BigInteger.ONE) > 0
40            || (n.gcd(z[i ])). compareTo(BigInteger.ONE) > 0)
            return false ;

42        // Save a_i in the array to hold values to be hashed.
        valuesToBeHashed[i + 2] = a[i ];

44    }
    // Check if the sum of e_i values are correct !
46    BigInteger e_hash = h.generateHash(valuesToBeHashed, t ). mod(pow.t);

48    if (e_hash.compareTo(e_sum) != 0)
        return false ;

50
    return true;

52 }

```

Listing A.8: Verification of a Vote

### A.3.2 Verification of Multiple Votes in a $t$ -out-of- $L$ Election

/\*\*

```

2 * The code for checking if MultipleVotes are of the correct form.
3 *
4 * @param mVotes The vote that will be checked
5 *
6 * @return true if the proof is correct.
7 */
8 public boolean checkMultipleVote(MultipleVotes mVotes) {
9     Vote[] votes = mVotes.getVotes();
10    BigInteger ea[] = mVotes.getEa();
11    BigInteger eb[] = mVotes.getEb();
12    BigInteger ec[] = mVotes.getEc();
13    BigInteger ed[] = mVotes.getEd();
14    BigInteger edb[] = mVotes.getEdb();
15    BigInteger f[] = mVotes.getF();
16    BigInteger e[] = mVotes.getE();
17    BigInteger z1[] = mVotes.getZ1();
18    BigInteger z2[] = mVotes.getZ2();
19    int clientId = mVotes.getClientId();
20
21    if (ea.length != noOfVotes || ea[0].length != noOfVotes
22        || eb.length != noOfVotes || eb[0].length != noOfVotes
23        || ed.length != noOfVotes || ed[0].length != noOfVotes
24        || edb.length != noOfVotes || edb[0].length != noOfVotes
25        || f.length != noOfVotes || f[0].length != noOfVotes
26        || z1.length != noOfVotes || z1[0].length != noOfVotes) {
27        return false;
28    }
29    int[] validVotesIndex = new int[votes.length];
30    for (int i = 0; i < votes.length; i++) {
31        if (checkProof(votes[i])) {
32            validVotesIndex[i] = 1;
33        } else {
34            validVotesIndex[i] = 0;
35        }
36    }
37
38    BigInteger[] valuesToBeHashed = new BigInteger[5];
39    valuesToBeHashed[0] = n;
40    valuesToBeHashed[4] = BigInteger.valueOf(clientId);
41
42    BigInteger voteDiff, temp1, temp2;
43
44    for (int i = 0; i < votes.length; i++) {
45        if (validVotesIndex[i] == 0) {
46            continue;
47        }
48        for (int j = i + 1; j < votes.length; j++) {
49            if (validVotesIndex[j] == 0) {
50                continue;
51            }
52

```

```

54
55
56         valuesToBeHashed[1] = ea[i][j];
57         valuesToBeHashed[2] = eb[i][j];
58         valuesToBeHashed[3] = ec[i][j];
59
60         e[i][j] = h.generateHash(valuesToBeHashed, t);
61
62         voteDiff = ((votes[i].getVote()).multiply(votes[j]
63             .getVote().modInverse(nArray[s + 1])))
64             .mod(nArray[s + 1]);
65         if (voteDiff.compareTo(ea[i][j]) != 0)
66             return false;
67
68         temp1 = ((ea[i][j].modPow(e[i][j], nArray[s + 1]))
69             .multiply(ed[i][j])).mod(nArray[s + 1]);
70         temp2 = encryptCheck(f[i][j], z1[i][j]);
71
72         if (temp1.compareTo(temp2) != 0)
73             return false;
74
75         temp1 = (edb[i][j].multiply(ec[i][j].modPow(e[i][j],
76             nArray[s + 1])).modInverse(nArray[s + 1]));
77         temp1 = temp1.multiply(eb[i][j].modPow(f[i][j], nArray[s + 1]))
78             .mod(nArray[s + 1]);
79         temp2 = encryptCheck(BigInteger.ZERO, z2[i][j]);
80
81         if (temp1.compareTo(temp2) != 0)
82             return false;
83     }
84 }
85 return true;
86 }

```

Listing A.9: Verification of *MultipleVotes*

### A.3.3 Verification of a *BinaryVote* in a 1-out-of-*L* Election

```

2  /**
3   * The code for checking if a BinaryVote is of the correct form.
4   *
5   * @param vote The vote that will be checked
6   *
7   * @return true if the proof is correct.
8   */
9  public boolean checkBinaryVote(BinaryVote vote) {
10     BigInteger[] encryption = vote.getEncryption();
11     BigInteger[][] e = vote.getE();

```

```

12     BigInteger [][] z = vote.getZ ();
13     BigInteger [][] a = vote.getA ();
14
15
16     // The array that holds the values to be used in the hash-function
17     BigInteger valuesToBeHashed[] = new BigInteger [4];
18
19     valuesToBeHashed[0] = n;
20     valuesToBeHashed[1] = BigInteger.valueOf(vote.getId ());
21
22     BigInteger u1,u2, voteCheckValue1, voteCheckValue2;
23
24
25     for (int j = 0; j <= 1; j++) {
26
27         // Check all e,a,z values to see if they are correct.
28
29         u1 = encryption[j].multiply(binaryVoteValues[j]);
30         voteCheckValue1 = a[0][j].multiply(
31             u1.modPow(e[0][j], nArray[s + 1])).mod(nArray[s + 1]);
32         voteCheckValue2 = z[0][j].modPow(nArray[s],
33             nArray[s + 1]);
34
35         if (voteCheckValue1.compareTo(voteCheckValue2) != 0)
36             return false ;
37
38         u2 = encryption[j].multiply(gInverse);
39         voteCheckValue1 = a[1][j].multiply(
40             u2.modPow(e[1][j], nArray[s + 1])).mod(nArray[s + 1]);
41         voteCheckValue2 = z[1][j].modPow(nArray[s], nArray[s + 1]);
42
43         if (voteCheckValue1.compareTo(voteCheckValue2) != 0)
44             return false ;
45
46         if ((n.gcd(a[0][j])).compareTo(BigInteger.ONE) > 0
47             || (n.gcd(z[0][j])).compareTo(BigInteger.ONE) > 0
48             || (n.gcd(a[1][j])).compareTo(BigInteger.ONE) > 0
49             || (n.gcd(z[1][j])).compareTo(BigInteger.ONE) > 0)
50             return false ;
51         // Save a.i in the array to hold values to be hashed.
52         valuesToBeHashed[2] = a[0][j];
53         valuesToBeHashed[3] = a[1][j];
54
55         // Check if the sum of e.i values are correct !
56
57         voteCheckValue1 = h.generateHash(valuesToBeHashed, t).mod(pow.t);
58
59         voteCheckValue2 = e[0][j].add(e[1][j]).mod(pow.t);
60
61         if (voteCheckValue1.compareTo(voteCheckValue2) != 0)
62             return false ;

```

```

64     }
65
66     // Now it's time to check the multiplication proofs.
67
68     BigInteger[] fi = vote.getFi();
69     BigInteger[] f = vote.getF();
70     BigInteger[] ed = vote.getEd();
71     BigInteger[] edb = vote.getEdb();
72     BigInteger[] z1 = vote.getZ1();
73     BigInteger[] z2 = vote.getZ2();
74     BigInteger[] eMult = new BigInteger[f.length];
75
76     // We set f[0] = e[0] to ensure that the calculations start from a
77     // correct value.
78     fi[0] = encryption[0];
79     BigInteger[] valuesToBeHashed2 = new BigInteger[5];
80     valuesToBeHashed2[0] = n;
81     valuesToBeHashed2[4] = BigInteger.valueOf(vote.getId());
82     for (int i = 0; i < 4; i++) {
83
84         valuesToBeHashed2[1] = fi[i];
85         valuesToBeHashed2[2] = encryption[i + 1];
86         valuesToBeHashed2[3] = fi[i + 1];
87         eMult[i] = h.generateHash(valuesToBeHashed2, t);
88         voteCheckValue1 = ((fi[i].modPow(eMult[i], nArray[s + 1]))
89             .multiply(ed[i])).mod(nArray[s + 1]);
90         voteCheckValue2 = encryptCheck(fi[i], z1[i]);
91
92         if (voteCheckValue1.compareTo(voteCheckValue2) != 0)
93             return false;
94
95         voteCheckValue1 = edb[i].multiply(fi[i + 1].modPow(eMult[i],
96             nArray[s + 1]));
97         voteCheckValue1 = voteCheckValue1.modInverse(nArray[s + 1]);
98         voteCheckValue1 = (voteCheckValue1.multiply(encryption[i + 1].modPow(fi[i],
99             nArray[s + 1]))).mod(n);
100         voteCheckValue2 = encryptCheck(BigInteger.ZERO, z2[i]).mod(n);
101
102         if (voteCheckValue1.compareTo(voteCheckValue2) != 0)
103             return false;
104     }
105
106     return true;
107 }

```

Listing A.10: Verification of a *BinaryVote*

### A.3.4 Verification of *MultipleBinaryVotes* in a *t*-out-of-*L* Election

```
/**
```

```

2 * The code for checking if MultipleBinaryVotes are of the correct form.
3 *
4 * @param mVotes The vote that will be checked
5 *
6 * @return true if the proof is correct.
7 */
8 public boolean checkMultipleBinaryVote( MultipleBinaryVotes mVotes ) {
9     BinaryVote[] votes = mVotes.getVotes ();
10    BigInteger ea [] = mVotes.getEa ();
11    BigInteger eb [] = mVotes.getEb ();
12    BigInteger ec [] = mVotes.getEc ();
13    BigInteger ed [] = mVotes.getEd ();
14    BigInteger edb [] = mVotes.getEdb ();
15    BigInteger f [] = mVotes.getF ();
16    BigInteger e [] = mVotes.getE ();
17    BigInteger z1 [] = mVotes.getZ1 ();
18    BigInteger z2 [] = mVotes.getZ2 ();
19    int clientId = mVotes.getClientId ();
20
21    if (ea.length != noOfVotes || ea[0].length != noOfVotes
22        || eb.length != noOfVotes || eb[0].length != noOfVotes
23        || ed.length != noOfVotes || ed[0].length != noOfVotes
24        || edb.length != noOfVotes || edb[0].length != noOfVotes
25        || f.length != noOfVotes || f[0].length != noOfVotes
26        || z1.length != noOfVotes || z1[0].length != noOfVotes) {
27        return false ;
28    }
29    int[] validVotesIndex = new int[ votes.length ];
30    for (int i = 0; i < votes.length; i++) {
31        if (checkBinaryVote(votes[i])) {
32            validVotesIndex[i] = 1;
33        } else {
34            validVotesIndex[i] = 0;
35        }
36    }
37
38    int l = votes[0].getFi ().length-1;
39    BigInteger[] valuesToBeHashed = new BigInteger[5];
40    valuesToBeHashed[0] = pk.getN ();
41    valuesToBeHashed[4] = BigInteger.valueOf( clientId );
42
43    BigInteger temp_, temp1, temp2, voteDiff;
44
45    for (int i = 0; i < votes.length; i++) {
46        // Skip the vote if it is not valid
47        if ( validVotesIndex[i] == 0) {
48            continue;
49        }
50        for (int j = i + 1; j < votes.length; j++) {
51            // Skip the vote if it is not valid
52            if ( validVotesIndex[j] == 0) {

```

```

54         }

55         valuesToBeHashed[1] = ea[i][j];
56         valuesToBeHashed[2] = eb[i][j];
57         valuesToBeHashed[3] = ec[i][j];

58         e[i][j] = h.generateHash(valuesToBeHashed, t);

59         temp_ = votes[j].getFi()[1]
60                 .modInverse(nArray[s + 1]);
61         voteDiff = ((votes[i].getFi()[1])
62                 .multiply(temp_)).mod(nArray[s + 1]);

63         if (voteDiff.compareTo(ea[i][j]) != 0)
64             return false;

65         temp1 = ((ea[i][j].modPow(e[i][j], nArray[s + 1]))
66                 .multiply(ed[i][j])).mod(nArray[s + 1]);
67         temp2 = encryptCheck(f[i][j], z1[i][j]);

68         if (temp1.compareTo(temp2) != 0)
69             return false;

70         temp1 = (edb[i][j].multiply(ec[i][j].modPow(e[i][j],
71                 nArray[s + 1])).modInverse(nArray[s + 1]);
72         temp1 = temp1.multiply(eb[i][j].modPow(f[i][j], nArray[s + 1]))
73                 .mod(nArray[s + 1]);
74         temp2 = encryptCheck(BigInteger.ZERO, z2[i][j]);

75         if (temp1.compareTo(temp2) != 0)
76             return false;

77     }

78     }
79     return true;
80 }
81
82 }
83
84 }
85
86 }
87
88 }
89
90 }

```

Listing A.11: Verification of *MultipleBinaryVotes*

### A.3.5 Verification of a *DecryptedShare* Acquired from a Decryption Server

```

/**
2 * This method verifies the zero-knowledge proof for the decrypted shares
3 * acquired from the decryption Servers
4 *
5 * @param ds
6 *     The DecryptedShare to be verified
7 * @return true if the DecryptedShare is correct.

```

#### A.4. The Combination of *DecryptedShares* Acquired from Decryption Servers

```
8  */
   public boolean checkShareDecryption(DecryptedShare ds) {
10      BigInteger [] vi = pk.getVi ();

12      BigInteger e = ds.getE ();
      BigInteger u = (ds.getC ().pow(4)).mod(nArray[s + 1]);
14      BigInteger u_tilde = (ds.getCi ().pow(2)).mod(nArray[s + 1]);

16      BigInteger uz = u.modPow(ds.getZ(), nArray[s + 1]);
      BigInteger u_tilde_e = u_tilde .modPow(e, nArray[s + 1]);
18      u_tilde_e = u_tilde_e .modInverse(nArray[s + 1]).mod(nArray[s + 1]);
      BigInteger check_1 = uz.multiply ( u_tilde_e ).mod(nArray[s + 1]);
20

      BigInteger v_tilde = vi[ds.getTallyId ()];
22      BigInteger vz = v.modPow(ds.getZ(), nArray[s + 1]);
      BigInteger v_tilde_e = v_tilde .modPow(e, nArray[s + 1]);
24      v_tilde_e = v_tilde_e .modInverse(nArray[s + 1]).mod(nArray[s + 1]);
      BigInteger check_2 = vz.multiply ( v_tilde_e ).mod(nArray[s + 1]);
26

      BigInteger [] valuesToBeHashed = new BigInteger[5];
28      valuesToBeHashed[0] = n;
      valuesToBeHashed[1] = check_1;
30      valuesToBeHashed[2] = check_2;
      valuesToBeHashed[3] = u;
32      valuesToBeHashed[4] = u_tilde ;

34      BigInteger digest = h.generateHash(valuesToBeHashed, ep.getT ());

36      if ( digest .compareTo(e) != 0)
          return false ;
38

      return true;
40
    }
```

Listing A.12: Verification of a *DecryptedShare*

#### A.4 The Combination of *DecryptedShares* Acquired from Decryption Servers

```
/**
2 * The method for combining the DecryptedShare's acquired from the
   * decryption Servers
4 *
   * @param dsList
6 *       Array of DecryptedShare's
   * @return The plaintext of the result of the election
8 * @throws InsufficientSharesException
   */
```

```

10 public BigInteger shareCombine(DecryptedShare[] dsList)
    throws InsufficientSharesException {
12     if (dsList.length < ep.getW()) {
        throw new InsufficientSharesException ("Not enough shares acquired");
14     }
    // Check ShareS!
16     BigInteger n = pk.getN();
    int needed = ep.getW();
18     int[] goodShares = new int[needed];
    int numberOfGoodShares = 0;
20     for (int i = 0; i < dsList.length; i++) {
        if (val.checkShareDecryption(dsList[i])) {
22             goodShares[numberOfGoodShares] = i;
            numberOfGoodShares++;
24             if (numberOfGoodShares >= needed)
                break;
26         }
    }
28     if (numberOfGoodShares + 1 < needed) {
        throw new InsufficientSharesException (
30         "Not enough good shares acquired");
    }
32     BigInteger lambda = BigInteger.ZERO;
    BigInteger c_ = BigInteger.ONE;
34     BigInteger delta = BigInteger.valueOf( Utility . faculty (ep.getl ()));
    // Lagrange interpolation
36     for (int i = 0; i < needed; i++) {
        lambda = BigInteger.ONE;
38         for (int j = 0; j < needed; j++) {
            if (j != i) {
40                 lambda = lambda.multiply(new BigInteger("-"
                    + (dsList[goodShares[j]].getTallyId () + 1)));
42             }
        }
44         lambda = lambda.multiply ( delta );
        for (int j = 0; j < needed; j++) {
46             if (j != i) {
                lambda = lambda
48                 .divide ((new BigInteger("
                    + (dsList[goodShares[i]].getTallyId () + 1)))
50                 .subtract (new BigInteger("
                    + (dsList[goodShares[j]]
52                     .getTallyId () + 1))));
            }
54         }
        c_ = (c_.multiply ( dsList [goodShares[i]].getCi ().modPow(
56             lambda.multiply (new BigInteger("2")), nArray[s + 1]))
            .mod(nArray[s + 1]));
58     }
60     // Applying algorithm for extracting the result !
    int s = ep.getS ();

```

#### A.4. The Combination of *DecryptedShares* Acquired from Decryption Service

```
62     BigInteger c_temp = c_;
63     BigInteger temp;
64
65     BigInteger [] lArray = new BigInteger[s + 1];
66     for (int i = 2; i <= s; i++) {
67         temp = c_temp.mod(n.pow(i));
68         temp = temp.subtract ( BigInteger.ONE);
69         lArray[i - 1] = temp.divide(n);
70         System.out. println (lArray[i - 1]);
71     }
72     temp = c_temp. subtract ( BigInteger.ONE);
73     lArray[s] = temp.divide(n);
74
75     BigInteger [] iArray = new BigInteger[s + 1];
76
77     iArray[0] = BigInteger.ZERO;
78
79     BigInteger counter = BigInteger.ONE;
80     BigInteger faculty [] = new BigInteger[s];
81     temp = BigInteger.ONE;
82
83     for (int i = 1; i < s; i++) {
84         counter = counter.add(BigInteger.ONE);
85         temp = temp.multiply ( counter);
86         faculty [i - 1] = (nArray[i]. multiply (temp.modInverse(nArray[s ])))
87             .mod(nArray[s]);
88     }
89
90     BigInteger t1, t2;
91     BigInteger i = BigInteger.ZERO;
92     for (int j = 1; j < s + 1; j++) {
93         t1 = lArray[j];
94         t2 = i;
95         for (int k = 2; k <= j; k++) {
96             i = i. subtract ( BigInteger.ONE);
97             t2 = (t2. multiply (i)). mod(nArray[j]);
98             t1 = (t1. subtract (n.pow(k - 1). multiply (
99                 t2. multiply ( BigInteger.valueOf( Utility . faculty (j))
100                     .modInverse(nArray[j ]))))). mod(nArray[j]);
101         }
102         i = t1;
103     }
104     temp = delta . multiply ( delta ). multiply (new BigInteger("4"));
105     temp = temp.modInverse(nArray[s]);
106     i = (i . multiply (temp)).mod(nArray[s]);
107     return i;
108 }
```

Listing A.13: Combination of *DecryptedShares* to a result



## Appendix B

# Digital Material following the Thesis

The source code for the electronic voting system, along with an executable version of all components of the voting system is included with this thesis. The source code, and executables for all packages used in the system are located in folders corresponding with their package names, in the *ElectionSystem* folder. The shell scripts used for starting the different components are located in the *ElectionSystem* folder.

The voting system and the simulator have been tested on Windows XP, and Ubuntu Linux, and cannot be guaranteed to run flawlessly on all systems. However, the program should work well on all computers that have Java Runtime Environment installed. As all components require the use of a graphical user interface, the computer used must use an operating system which supports graphical user interfaces

The Java documentation for all components is included in html in the *Electionsystem/doc* folder, and may be viewed by opening the *index.html* file in a browser.

