Cryptographic Voting Protocols

A Prototype Design and Implementation for University Elections at TU Darmstadt

Diplomarbeit

von

Alexander Klink

Betreut von Evangelos Karatsiolis

Technische Universität Darmstadt
Fachbereich Informatik
Fachgebiet Theoretische Informatik
Prof. Dr. Johannes Buchmann

März 2006

Acknowledgements

I would like to thank some people who were involved in the creation of this thesis:

- Berry Schoenmakers of TU Eindhoven for introducing me to the topic of cryptographic voting protocols.
- Evangelos Karatsiolis for his work in guiding me through the process of writing this thesis and his constant motivation.
- Roberto Samarone Araujo for hours of discussing various protocols with Evengelos and me.
- Warren D. Smith for providing with a preliminary version of his upcoming book »How Mathematics can Improve Democracy«.
- Ute Günther and Christian Burgmann for their proof-reading. Nonetheless, any remaing errors shall be blamed on me.
- Andrea Peter for her love and for cheering me up when I was down.
- Last but not least my parents, Jacqueline Herrnkind and Hans Peter Klink – without their love and continuous support, this thesis would never have been written.

Contents

List of Figures

Introduction

This thesis deals with cryptographic voting protocols and their application to university elections at TU Darmstadt. It is organized in two parts. The first one, the theoretical part, presents the state of research in the field of remote internet voting. Requirements for election protocols and several cryptographic algorithms for remote internet elections are presented. Based on the presented requirements one of the algorithms is chosen for implementation and analyzed under security aspects.

In the practical part, a prototypical implementation of this algorithm is presented as a proof of concept. This proof of concept is based on the conditions and background present at TU Darmstadt. Furthermore the security of this prototype will be considered which will lead to requirements for an implementation which might someday be used for university elections at TU Darmstadt.

The idea for this thesis came from the considerations about using online elections, which have been made at various committees within TU Darmstadt. Online elections promise to make the election procedures easier and to increase the turnout. Even though the turnout has skyrocketed from less than 10% to more than 40% during the last university election, online elections could still strengthen the participation. Due to these reasons, the *Hochschulversammlung* – TU Darmstadt's most important committee – decided on May 12th, 2004 to allow that absentee ballots can be cast electronically in the *Wahlordnung* (the document describing the voting process at TU Darmstadt). Because a security level that is similar to the one

available with »physical« elections was needed, the use of special cryptographic protocols should be considered. In connection with the decision to offer an infrastructure for »digital identities« for students at TU Darmstadt, the topic became more and more interesting. As I was interested in the topic not only professionally but also politically as a (in some cases former) member of different committees and student self-administration, I decided to write this thesis. In Prof. Dr. Johannes Buchmann I found an advisor who was interested in the topic professionally as well as politically, as he is also one of the vice presidents of TU Darmstadt.

During the last twenty years, quite some research has been done in the area of electronic voting systems. This research becomes more and more interesting with the use of more and more electronic voting systems all over the world – most prominently during the presidential elections in the United States. Unluckily, the systems used there are not based on cryptographic algorithms but are so called DRE (direct-recording electronic voting) machines, that just store the votes to be taken away physically for counting. As many problems and protests regarding these machines have been coming up lately (cf. [KSRW04], [Tru04]), a more »intelligent« solution would be desirable. Whether cryptographic algorithms combined with the state of security of todays personal computers are ripe enough to be used to conduct a large-scale election over the internet is highly debatable (and has actually been denied by various cryptographers, cf. [Rub], [JRSW04]). Still, it is an interesting area of research and some of the algorithms might be usable in a smaller setting, like in the proposed university elections.

Part I

Theoretical Part

1 Elections

It's not the voting that's democracy, it's the counting.

(Tom Stoppard)

1.1 Non-electronic elections

Elections are the instrument of decision-making in democracies. One crucial point of a fair election is the idea of the secret ballot. Even though one accepts it as *the* standard way of doing elections today, secret ballots have only been introduced in 1856 in Australia.

Nowadays, secret voting is even assumed to be a human right, as it is mentioned in the Universal Declaration of Human Rights in article 21:

> (3) The will of the people shall be the basis of the authority of government; this will shall be expressed in periodic and genuine elections which shall be by universal and equal suffrage and shall be held by secret vote or by equivalent free voting procedures.

1.2 Electronic elections

With the widespread use of elections in large countries and the introduction of more and more technology, methods to combine technology with

voting have been tried out and introduced. Starting from mechanical devices used for vote counting instead of a ballot marked on paper, more recently, so-called »electronic voting« has become more and more predominant. Electronic voting is the process of voting using specially created voting machines – normally in the form of a more or less ordinary computer with a touchscreen. These machines have rightly become the focus of criticism, as they normally offer very little protection against election fraud.

1.3 Elections using cryptography

A different idea that is starting to gain popularity is the possibility of remote voting (either using a standard personal computer or a special machine) over networks. Contrary to the voting machines, these schemes employ cryptography to ensure that election fraud is made (computationally) impossible.

As we will look more detailed into cryptographic voting schemes in the following, it makes sense to define who the participants of an election system are and what the requirements for such an online election would be.

Definition. Usually, the *participants* of an election system can be categorized as being one or more of the following:

- eligible *voters:* V_i for $i = 1, \ldots, k$ who are allowed to cast their ballot in the election,
- a tallier or multiple *talliers:* T_i, who are responsible for counting the votes,
- one or more *election authorities EA* (EA_i), which decide who is eligible to vote and performs supporting functions, as well as

- outside *observers*, who may view all or parts of the communication taking place during the election.

Definition. We call a voting system *secure* (largely following [BFP+01] here) if the following properties are satisfied:

- *Completeness:* if all participants are honest, the result is correct,
- *Soundness:* dishonest voters can not disrupt the voting,
- *Privacy* or *Anonymity:* all votes must be secret,
- *Unreusability:* no voter can vote twice,
- *Eligibility:* no one who is not allowed to vote can vote,
- *Fairness:* nothing must affect the voting, strongly connected to
- *Robustness:* the system tolerates (a certain amount of) cheating authorities,
- *Verifiability:* no one can falsify the result of the voting. There are actually two different kinds of verifiability:
  - *Universal* or *Public Verifiability:* every participant, possibly including outside observers, can verify that all votes have been counted correctly, and the weaker form of
  - *Private Verifiability:* every voter can check whether his vote was counted correctly but knows nothing about the correct tallying of votes not cast by himself.

Additional desirable properties of a voting system are *Receipt-Freeness* (no one can produce a receipt of how he voted) and *Uncoercibility* (no one can be forced to vote in a certain way). Receipt-Freeness can for example be achieved by deniable encryption, which enables the voter to lie about how he voted. As for uncoercibility, a system which allows the voter to change his vote until the last minute seems like an interesting solution idea

(which is actually implemented in several protocols), as this makes coercion not impossible, but still a lot harder. Yet unnoticed but still useful would be the idea to open the polling stations longer than allowing voters to vote electronically. So if »electronic coercion« takes place, the voter can still change his vote by voting physically in a better protected location.

1.4 Implementations of cryptographic voting protocols

Cryptographic voting protocols have unluckily been implemented and tried out in real-world scenarios quite seldomly. In the following, we will show some examples of elections that have been conducted (partially) using cryptographic voting protocols.

1.4.1 University elections

University elections provide a good chance to experiment with online elections, as they provide a relatively controlled setting with a small to medium number of voters (normally only a few thousand). Combined with the fact that most of the theory of electronic voting comes from within universities as well, this lead to quite a number of trial, shadow or real elections within the university setting.

Osnabrück 2000

The election of the student parliament of the University of Osnabrück in 2000 is probably one of the most interesting experiments in internet elections. The »Forschungsgruppe Internetwahlen« (research group internet

elections), who organized the election and wrote the software claims to have held the first legally binding online election over an open network. They used smart cards and blinded RSA signatures for the elections, in which more than 300 students participated.

All in all, the experiment went quite well (all legal challenges were dismissed by the electoral committee and none had to be resolved in court), but the report ([Int]) mentions some of the problems as well: smart card technology is not exactly user friendly and platform independent. Problems that could be solved theoretically but where not solved in time for the prototype include the secure use of the PCs by using a special operating system distributed on CD-ROM, connection analysis by providers by using a special election provider and anonymization to ensure long-term security.

The source of the developed software and the technical reports remain confidential.

1.4.2 Other official elections

Estonia 2005

Just recently, the local government council elections in Estonia made it into the international press. The president of the Republic of Estonia refused to sign an act which enables electronic voting (cf. [AR05]). He raises an interesting point in his criticism that voters who vote electronically and those who vote physically are not provided equal opportunities as the former are allowed to change their vote later (either electronically or physically). In the meantime, the election has successfully taken place. It used the already present public key infrastructure in Estonia but seems not to use any voting algorithm that is of particular theoretical interest.

2 Cryptographic Voting Protocols

God may not play dice with the universe, but something strange is going on with the prime numbers.

(Paul Erdős)

2.1 Cryptography

Cryptography is the science of information security. While, traditionally, cryptography was mostly used as a tool to en- and decrypt information using a secret key known to both parties (»symmetric cryptography«), modern cryptography has much more to offer:

- *Public key cryptography* can solve the key distribution problems by allowing two parties to communicate without sharing a secret key in advance.
- *Digital signatures* can be used to electronically sign documents. When implemented correctly, this is even as secure and legally binding as a physical signature.
- *Mix nets* and *onion routing* can be used to hide the fact that two persons are communicating and allow anonymous access to the Internet, for example.

- *Zero knowledge proofs* can be used to prove to someone that one knows a secret without revealing any information whatsoever about the secret itself.

- Last but not least, cryptographic primitives can be combined to create *secure electronic voting schemes* (cf. section 1.3 on what is meant when calling an electronic voting scheme »secure«).

Certain goals are commonly identified as the desired properties of cryptographic systems:

- *Confidentiality* – messages that are only meant to be read by a certain receiver (in cryptography, mostly Alice is talking to Bob and vice-versa) should only be readable to him and others should not be able to decrypt these messages.

- *Integrity* – it should be possible to protect messages from being tampered with while being sent over a communication channel. Even if it is sometimes impossible to protect against modification of the messages, the modification should be easily detectable.

- *Authentication* – it should be possible for the recipient to make sure that the sender is who he claims to be.

- *Non-repudiation* – the sender should not be able to deny that he sent the message.

Note that all of these goals will be needed for a successful voting scheme as well: ballots which are sent to an election authority electronically can only be read by the authority (or can rather be only tallied by an authority, as we don't want the authority to actually read the votes as well), ballots should not be tampered with while in transmission, only authorized voters should be able to vote and voters should not be able to deny that they voted at all.

For the mathematical background and the basic algorithms, readers are refered to [Buc04].

In the following, three different basic ideas for cryptographic voting protocols will be presented. In the first part, David Chaum's mix net concept will be presented along with several refinements for voting protocols. In the second part, voting using homomorphic encryption will be considered. Finally, the concept of blind digital signature and their relation to voting will be shown.

2.2 Voting using mix nets

Already in 1981, David Chaum was one of the first cryptographers who presented a special purpose cryptographic protocol for elections in [Cha81]. This protocol is based on untraceable mail using so-called »mix nets«, which will be presented here.

2.2.1 Mix nets

Mix nets use algorithms from public key cryptography to circumvent traffic analysis. They are successfully used in real-world applications, for example in anonymous remailers (so called »mixmaster« remailers) or with the JAP (Java Anonymous Proxy) project. A similar, but slighty different approach is implemented in so-called onion routing protocols, as for example implemented in Freenet and TOR (The Onion Router) to allow anonymous communication and internet usage.

Basic definitions from public key cryptography used in mix nets are presented below.

Definition. To *encrypt* or *seal* a message X using a public key K, the user calculates $K(R,X)$, where $K(X)$ denotes the encryption of X under an encryption algorithm with the key K. Here, R is a random bitstring which is prefixed to the message to avoid the problem that the message can be guessed by trying out different plain text messages Y and testing whether $K(X) = K(Y)$.

Definition. To *sign* a message X, the user calculates $Y = K^{-1}(C,X)$, where K^{-1} is the private key corresponding to the public key K (or rather the encryption function under K^{-1} in this context).

Here, C is a large constant (e.g. a bitstring containing all zeroes). Anyone in possession of the public key can then calculate $K(Y) = C,X$ and check the constant C.

The idea of mix nets is based on two important *assumptions* (following [Cha81]):

1. No one can determine anything about the correspondences between a set of sealed items and the corresponding set of unsealed items, or create forgeries without the appropriate random string or private key.
2. Anyone may learn the origin, destination(s), and representation of the messages in the underlying communication system and anyone may inject, remove or modify messages.

The mail system

The people in the crypto system who wish to communicate with each other will be aided by a computer called »mix«. If a user, let's call her Alice (or shortly A) wants to send an encrypted message to another user, say Bob (B), without revealing that she communicates with him, she has to

include the mix in the communication. Therefore, she encrypts the message M_B with Bob's public key K_B, adds the address B and encrypts the result with the public key K_1 of the mix. As a formula, her message to the mix looks as follows: $K_1(R_1, K_B(R_0, M_B), B)$. The mix now receives this message, decrypts it and throws away the random string R_1, it remains $K_B(R_0, M_B), B$.

When the mix has received enough messages, it can reorder them (for example randomly or in lexicographical order) to hide the correspondence between the input and output of its messages (that is the reason for calling it »mix«) and send this whole batch to the corresponding recipients (cf. figure 2.1 for a small example).

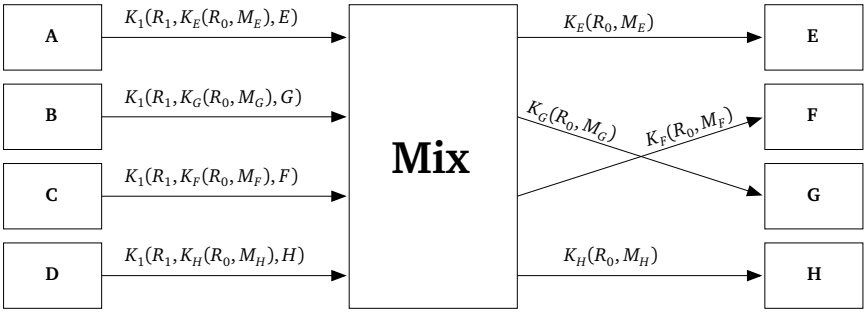

Figure 2.1: A small mix net and the corresponding communication

By assumption 1, no cryptographic attack can be used to see a relationship between the encrypted input messages and the decrypted output messages of the mix. But this assumptions only holds when messages are not repeated. Assume that Alice sends the same message $K_1(R_1, K_B(R_0, M_B), B)$ addressed to Bob to the mix twice, then the mix sends the same output message $K_B(R_0, M_B)$ to Bob twice. An eavesdropper can thus deduce that Alice and Bob communicate with each other. Fortunately, this problem can be solved easily, for example by inserting a time-dependent value into the

random bitstrings to make sure that they do not repeat. Alternatively, the mix can remember for the time it uses the same public key, which messages it has already sent and not forward any duplicates.

An additional nice function of the mix can be implemented by having the mix sign a receipt for Alice that it has received a message, which would look like this for a message to Bob:

$$Y = K_1^{-1}(C, K_1(R_1, K_B(R_0, M_B), B))$$

Then every participant can prove when the mix does not forward his message by showing the receipt Y, the missing message $X = K_B(R_0, M_B), B$ and the random string R_1 such that $K_1(Y) = C, K_1(R_1, X)$. When the mix signs the complete batch of messages, it can thus be shown with the batch and the corresponding signature that a certain message was not sent.

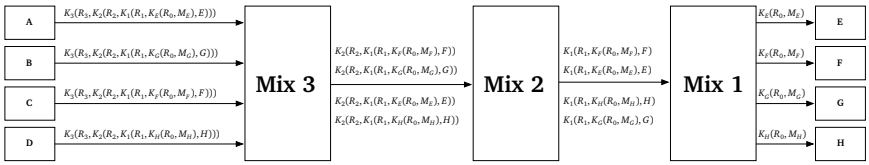

Figure 2.2: A cascade of three mix servers

To increase the security of mix nets, several mixes can be used consecutively in a *cascade* (cf. figure 2.2). In this case, we only need the signed receipt of the first mix, because the mixes later in the cascade can show that a message was not in their input with the signed output of their predecessor. A message for n mixes is prepared in the same way as for one mix and then sealed step-by-step with the public keys of the mixes, the first mix receives a message like this:

$$K_n(R_n, K_{n-1}(R_{n-1}, \ldots, K_2(R_2, K_1(R_1, K_B(R_0, M_B), B))\cdots))$$

After peeling of the topmost layer, it forwards a batch of messages of the form

$$K_{n-1}(R_{n-1}, \ldots, K_2(R_2, K_1(R_1, K_B(R_0, M_B), B))\cdots))$$

The last mix outputs a batch of messages of the form $K_B(R_0, M_B)$ to the corresponding recipients, just as in the case of only one mix.

2.2.2 Digital pseudonyms and Chaum's voting protocol

Using this mail system, any message can now be sent anonymously, including public keys. A *digital pseudonym* is a public key used to verify signatures of the anonymous owner of the corresponding private key. A *roster* is a list of pseudonyms created by an authority. This authority decides, which applications for addition to the list are accepted but it can not trace the pseudonyms in the completed roster.

The applications are sent to the authority anonymously and contain (in addition to the data needed for the decision whether to include the pseudonym in the roster or not) a special unaddressed message. This message is sent through a mix or a cascade of mixes by the authority at the end of the application process. It is of the form $K_1(R_1, K)$ for one mix or $K_n(R_n, \ldots, K_2(R_2, K_1(R_1, K))\cdots)$ for n mixes, i.e., it contains the digital pseudonym as a message. Once all applicants are verified, the authority sends the messages of all the applicants who are to be listed in the roster

through the mix or the mixes. The output of the last mix is thus a list of digital pseudonyms accepted by the authority – the roster.

By only enlisting registered voters into the roster, we can construct a voting protocol. If there is only one mix, the voter sends the his ballot as the message $K_1(R_1, K, K^{-1}(C, V))$ to the mix. If a cascade of n mixes is used, the message looks like this:

$$K_n(R_n, \ldots, K_2(R_2, K_1(R_1, K, K^{-1}(C, V)))\cdots)$$

Here, K is the pseudonym of the voter and V is the ballot. The output of the final mix is a batch of the form $K, K^{-1}(C, V)$ (possibly ordered lexicographically). Because the roster of the pseudonyms of the registered voters is also ordered lexicographically by K, the lists can be checked simultaneously. A ballot is only counted when the pseudonym K is in the roster and the signature of the ballot has been verified successfully.

2.2.3 Further reading

Chaum's basic voting protocol and the technique behind mix nets have been refined by various cryptographers, some papers in this area are: [Wik04], [BG02], [JJ99], [MH96] and [Nef04].

2.3 Voting using homomorphic encryption

2.3.1 Homomorphic encryption

Definition. An encryption function E is called *homomorphic*, if for some operations $\oplus, \otimes$, the following holds:

$$E(x \oplus y) = E(x) \otimes E(y)$$

If the operations $\oplus, \otimes$ are in fact our »normal« plus and times, this is especially useful for tallying voting results. Say each voter is only able to vote 0 for no and 1 for yes, than one can multiply the encrypted ballots to receive a product of encrypted ballots which is equal to the encryption of the sum of the ballots, i.e., $\prod_i^n E(b_i) = E(\sum_i^n b_i)$. Thus, the final tally can be computed without looking at the individual votes because of the homomorphism property of the encryption function. If the key necessary for decryption is shared between several keyholders, this provides a good protection against compromising individual votes.

2.3.2 Smith's voting scheme using homomorphic encryption and zero knowledge proofs

In [Smi05], Warren D. Smith presents a voting scheme that fulfills all the properties for a secure voting scheme and offers receipt-freeness as well as a certain amount of protection against coercion by allowing the voter to vote multiple times (with only the last vote counted). His scheme employs homomorphic ElGamal encryption and uses quite a number of zero-knowledge-proofs to provide confidence in the correctness of the ballots and the counting.

Here is how the basic protocol works:

Preparation

1. The s talliers randomly create their secret partial decryption keys $K_1, \ldots, K_s$ and use them to produce the public encryption key $(g, h = g^K)$ for the election authority (EA), where $h = g^{K_1} \cdot g^{K_2} \cdots g^{K_s}$, i.e., $K = K_1 + \cdots + K_j$.

2. The EA publishes g, h and i, where i is a random group element used for encryption later (it is used to satisfy that the encryption function is homomorphic). Note that it does not know the corresponding decryption key K as this key is shared by all talliers. Furthermore, for creating signatures, the EA creates a special signature-only key for which it knows the decryption key (as this is necessary to create the signatures). The EA publishes the public key for this signature key as well (for verification purposes).

3. The EA sets up a world-readable bulletin board with a designated space for each registered voter.

Sending the ballot

4. The voter generates his vote v (say 0 or 1 for yes or no), ElGamal-encrypts i^v with K and sends the resulting message $M = (g^r, h^r i^v)$ to the EA.

5. The EA ElGamal-reencrypts the message by multiplying with g^q, where q is chosen randomly. It sends the resulting $M' = (g^{r+q}, h^{r+q} i^v)$ back to the voter. Note that this is basically the same as multiplying the message with an encryption of 1.

6. The voter and the EA jointly provide a zero-knowledge proof P_1 that the re-encrypted vote is valid (for example by showing that either $i^0 = 1$ or $i^1 = i$ was encrypted – but without revealing any information about the vote).

7. If this proof is invalid, the EA alerts the voter of the fact and does not accept the vote. Note that if one has a »rogue« EA, it might not accept the vote at all at any point. Help against this comes only from having trust in the EA or distributing the work of the EA among different entities (which is easily possible using this protocol).

8. The EA creates a designated-verifier zero-knowledge proof P_2 for the voter that proves that M and M' encrypt the same vote. This can be shown by proving the logical statement (»I know the value of q« or »I know the private key of V«). As this proof is designated-verifier, it is of no use to convince anyone of the correct re-encryption except for the voter. This is quite useful as thus the voter can not show to anyone how he voted. Together with the proof, the EA sends the date D that will be used on the bulletin board to show when a voter voted.

9. The voter checks the validity of both the date and the re-encryption proof. If both are valid, the voter signs (M', P, D) and sends it back to the EA.

10. If the voter's signature is invalid, the EA alerts the user of the fact.

11. EA signs (M', P, D), too.

12. The EA posts the encrypted ballot M', the signatures of both the voter and the EA, the zero-knowledge proof of validity as well as the date next to the voter's name on the bulletin board. For a better physical security, both the voter and the EA can print a copy of this information, too (for example encoded as a barcode).

13. Optionally, the voter may scan the receipt and check that the signatures and the proof are recorded correctly.

14. Optionally, the voter may vote again (maybe differently because he was coerced into the first vote), in the end, only the last dated vote counts.

Tallying

15. The EA (or somebody else, maybe the talliers together so that no one can cheat and only use a subset of the votes) multiplies the encrypted

votes (of course only the most-recently-dated, valid and signed votes) to get an encryption of i raised to the sum of the votes – the tally T.

16. The talliers successively partially decrypt the totals using $K_1, \ldots, K_s$. They provide a zero-knowledge proof that they are using the same keys that were used for the joint generation of K.
17. Once they are finished, they broadcast i^T.
18. Somebody (the one with the greatest computing power) uses the baby step/giant step or for example the Pollard-λ method to determine the discrete logarithm T and broadcasts it. Note that the correctness of T can be immediately verified by computing i^T again.

Security considerations

Let's have a look at why this scheme provides the security features we introduced in section 1.3.

- *Completeness:* If both the voters and the EA are honest and follow the given protocol, all votes will be valid and can be counted correctly. The correctness of the tally is based on the homomorphic property of the protocol.
- *Soundness:* Dishonest voters would either like to submit a ballot with a vote encoding multiple »no« or »yes«-votes – in our case the v_i would then be either be a negative or a positive number different from 1. This is infeasible for the voter, as he would have to fake the corresponding zero knowledge proof in step 6 – which is computationally impossible.
- *Privacy:* The privacy of the protocol lies in the distributed nature of the EA. The private key needed for the decryption of the single votes is shared among entities who distrust each other. This allows for some »social security« that they would not want to cooperate to

decrypt the single votes but rather only work together to decrypt the tally.

- *Unreusability:* As every voter signs his vote that is published as the final encrypted tally and the multiplication of the relevant votes can be verified by anyone, no voter can vote twice.
- *Eligibility:* To be able to vote without being a registered voter in this scheme is computationally impossible, as one would have to fake ElGamal signatures of someone who is a registered voter.
- *Fairness & Robustness:* If the election authorities are chosen in such a way that they will not collaborate on any voting fraud activities, these features are reached as well. If the sub-keys of the single authority organizations are again (secret-)shared within the organization, one can reach a certain robustness against a few cheating persons, who for example will not agree to begin tallying.
- *Universal Verifiability:* Outside observers can verify that the votes have been registered correctly, as they can view the public bulletin board. They can also check that the multiplications of the ballots are correct.
- *Private Verifiability:* Every voter can check that his vote has been accepted as he can check whether it was correctly posted on the bulletin board. If it is not present on the bulletin board, but has been accepted by the EA, it is actually signed by the EA, so the voter can show to an outside observer that something is going wrong. If the voter keeps this information in an offline storage medium, for example on paper, he can even prove this without the help of data stored on his (potentially vulnerable) computer.
- *Receipt-Freeness:* As the vote that is posted on the bulletin board is a re-encryption of the voter's actual encrypted vote, the voter can not show that he voted in a certain way to someone outside the system. If only the normal encryption of the vote had been used, the

voter could have simply opened his vote and his random number to the outsider to prove that he voted in a certain way. With the re-encryption, he does not know the new random number that has been used to encrypt his vote and is thus unable to produce a receipt that can be trusted. On the other hand, the voter can be sure that the re-encryption is correct, as he has been provided with a designated-verifier zero knowledge proof of that fact by the election authority. This designated-verifier zero knowledge proof is only interesting to him, if he gives it away, the outsider already knows that the right side of the OR-statement (»I know the private key of V«) is correct, which leaves him doubtful about the correctness of the left hand side – it could either be true or false and the proof would still be valid.

- *Uncoercibility:* As voting multiple times is possible, this provides a certain level of protection against coercion. Of course, receipt-freeness and uncoercibility are closely related, as without physical presence of a coercer, a receipt that someone has voted in a certain way would have to be produced, which is clearly impossible as seen in the last paragraph

2.3.3 Smith's voting scheme adapted to the TUD background

Given the good security features and the possibility that this protocol can in fact be implemented – which is not easy to see for many of the more theoretical papers in the area of cryptographic voting protocols, Smith's voting scheme was chosen as the candidate for a prototype implementation based on the situation at TU Darmstadt.

This requires some (minor) modifications to the protocol, as it only shows how to vote once in a yes/no-situation Here is a description of an adaption to the situation at TU Darmstadt, which allows a voter to vote for a committee, i.e., for k out of n candidates or lists. Compare figure 2.3 for an

overview of the communication between voter and election authority that takes place during the voting phase.

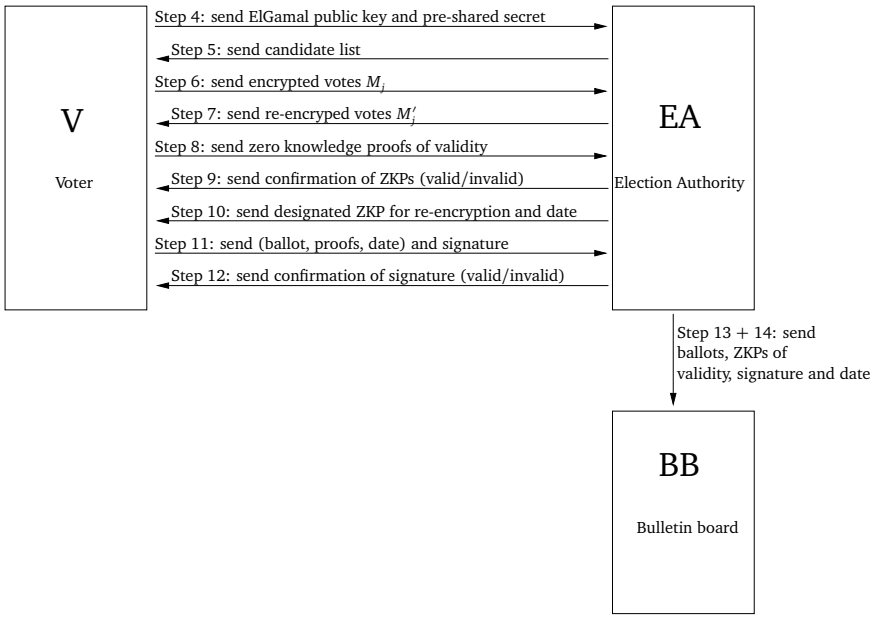

Figure 2.3: Communication during the voting phase

Preparation

1. The s talliers randomly create their secret partial decryption keys $K_1, \ldots, K_s$ and use them to produce the public encryption key $(g, h = g^K)$ for the election authority (EA), where $h = g^{K_1} \cdot g^{K_2} \cdots g^{K_s}$, i.e., $K = K_1 + \cdots + K_j$.
2. The EA publishes g, h and i, where i is a random group element used for encryption later (it is used to satisfy that the encryption function is homomorphic). Note that it does not know the corresponding decryption key K as this key is shared by all talliers. Furthermore, for creating signatures, the EA creates a special signature-only key for which it knows the decryption key (as this is necessary to create the signatures). The EA publishes the public key for this signature key as well (for verification purposes).
3. The EA sets up a world-readable bulletin board with a designated space for each registered voter. The EA (as usual) publishes a list of eligible voters and which committees they are allowed to vote for. This list can be appealed by the voters and changed until a certain date at which it is fixed (the process is the same as in the usual physical election at TUD except for that it may now be done – at least partially – electronically).

Sending the ballot

4. The voter creates an ElGamal key and sends the signed public key to the EA, who adds the key to its directory if the signature is valid. Note that the key has to be signed with a known public key (for example the RSA key available on the TUD smart card). Alternatively, the voter can send his public key together with a pre-shared secret to the EA (for that, printing a long random string onto the »Wahlbenachrichtigungen« which are sent to each voter physically would be a

reasonable idea) and the EA enters him into its directory if the secret is correct.

5. The EA sends the voter a list of candidates for the first committee for which he is eligible to vote. Say it contains n candidates and the voter should be able to vote for a maximum number of $k \leq n$ candidates.

6. For each candidate, the voter generates his vote v_j (say 0 or 1 for yes or no), ElGamal-encrypts i^{v_j} with K and sends the resulting messages $B = (M_1, \ldots, M_n)$ to the EA where $M_j = (g^r, h^r i^{v_j})$.

7. The EA ElGamal-reencrypts the messages by multiplying them with g^{q_j} and h^{q_j} respectively, where q_j is chosen randomly. It sends the resulting $B' = (M'_1, \ldots, M'_n)$ with $M' = (g^{r+q_j}, h^{r+q_j} \cdot i^{v_j})$ back to the voter.

8. The voter and the EA jointly provide zero-knowledge proofs P_j that the re-encrypted votes are valid, by showing that it is an ElGamal-encryption of either $i^0 = 1$ or $i^1 = i$. Additionally, they prove that the product of the re-encrypted votes $M'_1 \cdots M'_n$ is an ElGamal encryption of a number in the set $\{i^0, i^1, \ldots, i^{k-1}, i^k\}$ (thus proving that at most k »yes«-votes were cast). Let's call this proof P_{int}.

9. If any of these proofs are invalid, the EA alerts the voter of the fact and does not accept the vote.

10. The EA creates designated-verifier zero-knowledge proofs Q_j for the voter that prove that M_j and M'_j encrypt the same vote. This can be shown by zero knowledge proving the logical statement (»I know the value of q_j« or »I know the private key of V«). As this proof is designated-verifier, it is of no use to convince anyone of the correct re-encryption except for the voter. This is quite useful as thus, the voter can not show to anyone how he voted. Together with the proof, the EA sends the date D that will be used on the bulletin board to show when a voter voted.

11. The voter checks the validity of both the date and the re-encryption proof. If both are valid, the voter signs $(B', (P_1, \ldots, P_n, P_{int}), D)$ and sends it back to the EA.

12. If the voter's signature is invalid, the EA alerts the user of the fact.

13. EA signs $(B', (P_1, \ldots, P_n, P_{int}), D)$, too.

14. The EA posts the re-encrypted ballot B', the signatures of both the voter and the EA, the zero-knowledge proofs of validity as well as the date next to the voter's name on the bulletin board. For a better physical security, both the voter and the EA can print a copy of this information, too (for example encoded as a barcode).

15. Optionally, the voter may scan the receipt and check that the signatures and the proof are recorded correctly.

16. Optionally, the voter may vote again (maybe differently because he was coerced into the first vote), in the end, only the last dated vote counts.

Tallying

17. The EA (or somebody else, maybe the talliers together so that no one can cheat and only use a subset of the votes) multiplies the encrypted votes for each candidate (of course only the most-recently-dated, valid and signed votes) to get an encryption of the sum of the votes i^{T_j}, where T_j corresponds to the tally for candidate j.

18. The talliers successively partially decrypt the totals using $K_1, \ldots, K_s$. They provide a zero-knowledge proof that they are using the same keys that were used for the joint generation of K.

19. Once they are finished, they broadcast the i^{T_j}.

20. Somebody (the one with the greatest computing power) determines the discrete logarithms T_j and broadcasts them. Note that the correctness of T_j can be immediately verified by computing i^{T_j} again. Tests on a standard Intel Celeron 1.7 GHz PC show that one can easily calculate 700 exponentiations per second without any optimization. Thus, using advanced discrete logarithm is barely necessary, as the necessary calculation could be done in a few seconds. The worst case would be that all voters voted yes, which would mean that one would have to try out about 20000 exponentiations in the application scenario, which would take around 30 seconds. If one would like to compute the next tally even faster, one could save the computed values and compare the results against them – for a university with 20000 voters, this would even be possible in RAM.

2.3.4 Further reading

Homomorphic encryption has been used as an election system by many different people, some papers in this area are: [CGS97], [Sch99], [CFSY95], [BFP+01], [LK02], [LK00], [Acq04], [DGS03], [KY04] and [Ben96].

2.4 Voting using blind signatures

Behind the idea of so-called »blind signatures« is once again David Chaum, who published a paper called »Blind signatures for untraceable payments« ([Cha82]) in 1982.

The basic idea of blind signatures can be described by a real-world analogy using envelopes and carbon paper. Chaum describes the »real-world version« in his paper already with an application to voting: Say a trustee wants to hold an election but the voters can not meet in person to put the ballots into a ballot box. This can be solved by the following protocol: The

voter places a ballot slip with the vote written on it in a carbon lined envelope, puts the envelope itself into another envelope and mails it to the trustee. The trustee then opens the outer envelope, signs the inner envelope so that the signature ends up on the ballot slip (because of the carbon paper the inner envelope is made of). He puts the inner envelope into a new envelope and mails it back to the voter. The voter opens all envelopes and puts the signed slip of paper into a new envelope and mails it anonymously to the trustee. The ballots received by the trustee can be put on public display, i.e., they can be counted publicly. If the voters remember some characteristic of their ballot slip, for example the kind of paper used, they can make sure that their vote was really counted.

Several algorithms to put the idea behind the physical process mentioned above into cryptography have been proposed.

2.4.1 Blind signatures using Nyberg-Rueppel

The Nyberg-Rueppel signature scheme uses ElGamal as well and works as follows (presented using the notation used in [Smi05]). Let $P = 2Q + 1$ and Q be large primes. Furthermore, let g be a nontrivial square mod P. All of P,Q and g are public information. Alice now (as in the original ElGamal algorithm) publishes $h = g^K$ and keeps K secret. If she wants to sign a message M mod P now, she creates a random z mod Q and computes $r = Mg^z$ mod P and $s = Kr + z$ mod Q. The pair (r,s) is then available as the signature of M. Anybody may verify the signature by checking the identity $M = g^{-s}h^r r$.

To use this scheme in a blinded way, let g and $h = g^K$ be as above, with h being the public and K being the private key of a trustee, let's call him Tom. He then creates a random integer γ mod G, computes $I = g^\gamma$ and sends it to Alice. Alice then creates random integers α and β mod G, computes $r = Mg^\alpha I^\beta$ mod G and $B = r/\beta$ mod G and sends B to Tom. Tom computes $S = KB + \gamma$ mod G and sends it back to Alice, who then

computes $U = S\beta + \alpha \bmod G$. (r, U) is then Tom's signature of Alice's message M.

2.4.2 Blind signatures and mix nets

In [FOO92], Fujioka, Okamoto and Ohta describe a voting scheme that makes use of both blind signatures and a mix net, which is used as an anonymous channel.

The outline of the scheme is as follows:

1. The voter fills in a ballot and (using a digital signature for authentication) gets a blind signature from an election authority *EA*.
2. At the end of the administration stage, the election authority publishes the list of received blinded ballots.
3. The voter sends the ballot to the tallier via an anonymous channel (e.g. a mix net).
4. At the end of the election, the voter sends his encryption key via the anonymous channel to the tallier.
5. The tallier decrypts the ballots and counts and publishes the result

Security considerations

Regarding the security requirements outlined in chapter 1, the authors note that sending an illegal key which can not open the vote causes a problem with *soundness*, as the outside observer can not distinguish whether this was caused by a dishonest voter or by a dishonest tallier. As a solution, the authors suggest that the voter should send his key to more parties (for example the candidates). This of course increases the complexity, as for each receiver, another anonymous channel will be needed.

The *anonymity* of the scheme depends on (as the name suggests) the security of the anonymous channel. If the anonymity of the channel is broken, the vote can be traced to the voter. With regard to anonymity, the scheme thus provides no more security than any other scheme using mix nets.

Conclusions

A disadvantage of this scheme is that it requires interaction of the voter in different stages of the protocol (administration, ballot sending, opening).

2.4.3 Farnel Protocol

The Farnel protocol is a protocol based on the physical elections in Brazil developed by Roberto Samarone Araujo, which was discussed by him, Evangelos Karatsiolis and myself. In the beginning of our brainstorming sessions on this protocol, it looked like this:

Here, the steps executed (as indicated in the diagram) are as follows

1. Voter and Voting Authority authenticate each other
2. The VA sends an empty ballot to the scrutineers
3. The scrutineers sign the empty ballot and send it back to the VA
4. The VA sends the signed empty ballot to the voter V
5. The voter makes his votes, makes a blind signature on it and sends it to the VA
6. The VA signs the blinded ballot and passes it on to the scrutineers

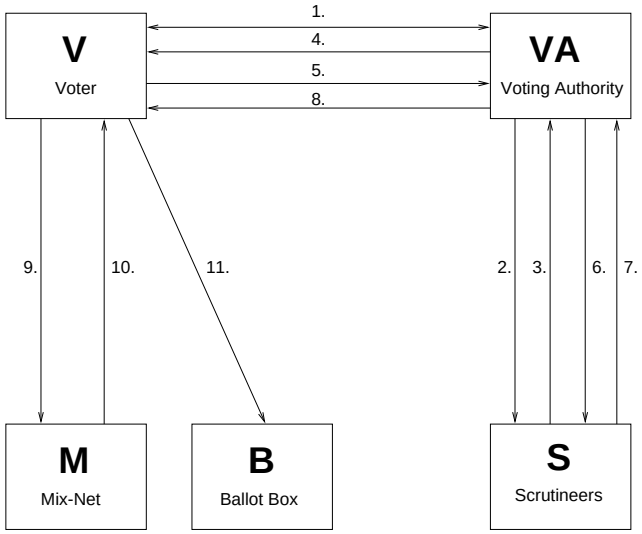

Figure 2.4: The Farnel protocol

7. The scrutineers checks the signature and send the ballot back to the VA
8. The VA passes the ballot back to the voter
9. The voter unblinds the ballot and sends it into the mixnet
10. The mixnet sends a random ballot to the voter
11. The voter signs the random ballot and forwards it to the ballot box server

During the discussion of this protocol, we realized that steps two and three can be omitted, since the signed empty ballot is always of the same form, so letting the scrutineers sign the ballot is not a security feature and does not help against the duplication of an empty ballot.

A question arose whether we want the voter to send the blinded ballot together with a proof that the vote that is contained is well-formed (not to be confused with valid, because we still want the voter to be able to vote invalid). I still believe this is not a technical but a political decision whether to have write-in ballots. At the moment, in physical elections, write-in ballots are possible but are not published. As the contents of the ballot box might be posted publicly on the internet, the situation here is of course a bit different.

The idea that is behind step ten and eleven comes from the physical world, too. Here, the analogy would be putting a vote into a first ballot box and taking out another random vote that is then put into the second ballot box. In the electronic version, it has the problem that the voter might not put the vote he's been sent into the second ballot box (as he might suspect that it is better for him not to put it there, for example if a large number of voters vote for a party the voter does not want to support). This problem can probably be solved by only deleting the ballot from the mix net once it has entered the ballot box.

Another problem in the first version was, that in step 11, the voter could send his original vote instead of the vote that he was sent, so the vote sent in the step 10 would have to be signed by the mixnet so that only votes signed by the scrutineers, the mixnet and a voter would be counted in the second ballot box.

The protocol also still suffers from the *ballot box problem*: the ballot box in this protocol has to be trusted ultimately as it can delete any vote and claim to have never received it. This problem could probably be solved by using multiple ballot box servers so that one can do a majority decision on what the correct ballot is (if less than a majority of servers collude).

Unluckily, the deficiencies of the protocol could not be solved without completely changing it, so it servers mostly as an example on how many things you have to consider during a protocol design. Considering that it

took a while to find the shortcomings, it is surprising that – to my knowledge – none of the cryptographic voting protocols have been proven correct.

2.4.4 Further reading

Blind signature voting protocols have been proposed in various forms, some papers in this area are [FOO92], [JL97], [RRN01], [Oka97], [CC97] and [Lee99].

3 Traditional elections versus cryptographic election systems

> Tradition means giving votes to the most obscure of all classes – our ancestors. It is the democracy of the dead. Tradition refuses to submit to the small and arrogant oligarchy of those who merely happen to be walking around.
>
> *(Gilbert K. Chesterton)*

In the following, we will set out to compare »traditional« elections and cryptographic election systems. We will see that both have their respective shortcomings.

3.1 Problems that exist only in traditional elections

3.1.1 The »greek ballot envelope problem«

In elections in Greece, the ballots are put into special signed envelopes (as they are in university elections at TU Darmstadt, too). If one now has a chance to sneak away a single envelope and ballot paper from the election (for example by submitting the ballot into the ballot box in a normal envelope while the election helpers are not concentrated enough to spot it), this ballot can be used for vote-buying. You can then make a vote in presence of someone who pays you money for it, put it into the envelope and seal it shut. Once you enter the voting booth, you can take away the envelope given to you and the ballot paper. After having presented this to the coercer, he can be sure you voted for him. Note that the envelope and ballot paper can be used to repeat this process with another voter. This problem has actually occured in the wild in Greece and has no analogy in the electronic world – if something similar to an envelope is used in an electronic voting scheme, it is created for each user at the time of voting. Note that this problem could be easily prevented by making it impossible or forbidden to seal the envelopes shut.

3.1.2 Ballot marking

Determining whether a ballot is marked physically is quite a problem – is a »hook« instead of a cross a mark which invalidates the ballot? To discourage coercion and vote-selling, the symbol that makes a ballot a valid vote should always be exactly the same, which is physically impossible. Digital yes/no-votes can be encoded in binary 0/1, which always »look« exactly the same and can not be made in such a way that they look different.

3.1.3 Fairer voting methods are possible

There exists mathematical models which show that there are fairer voting methods available then the ones used today. Unluckily, most of them use a more complicated way of making a ballot as well, for example choosing a number in a certain range. This makes tallying by hand impossible. Using electronic voting, tallying methods that are different from just summing up the ballots can easily be implemented.

3.1.4 Disabilities

Providing a fair election experience for people with physical disabilities proves to be quite a hard problem. Interestingly enough, the final push for the adoption of direct recording electronic (DRE) voting machines in the United States of America was the Help America Vote Act (HAVA) of 2002, which enforced and supported the study of accessibility in electronic voting machines.

Better privacy can be provided using computers for example for blind voters using headphones then with traditional methods in physical elections.

3.2 Problems that exist only in electronic elections

3.2.1 Transfer of authentication information

The problem of transfer of authentication information, e.g. selling the private key used in the voting process, is much larger in electronic elections than in traditional elections. Because there is no physical verification of the presence of the voter at a voting booth, keys might be sold and used by

someone who is not the voter. Smart cards help a bit in this regard, as they are able to prohibit the transmission of the private key via networking, but of course they can be physically sold and mailed as well.

This problem is inherent in any kind of remote, unverified voting, there is no good solution that allows both convenience (voting from your home) and security.

As for the university elections, I do not consider this as a large problem, as identity checking is only based on the election notification, who can be easily transferred to somebody else as well. Of course that might allow you to vote maybe 5-10 times or so until the election judges get suspicious, whereas in the electronic case, it is possible to do this as often as possible without detection.

3.2.2 Long-term ballot secrecy

In correctly performed paper-based elections, the secrecy of the ballot is protected perfectly after the election, as there exists no record of who voted how.

All the cryptographic protocols presented in this thesis only provide computational security by their very nature. In some cases, the encrypted ballot is only transmitted to a voting authority, in others it is stored on a public bulletin board for all to see. But regardless of these differences in »publicness« it seems reasonable to assume that an attacker could get hold of all cryptograms.

Given the fact that every cryptosystem invented has been broken over time, it seems highly possible that the typical cryptographic algorithms used in the voting schemes will be broken someday – either by mathematical or technological advances.

It is then a political and sociological problem, how much the ballots of an election several years ago (10? 20? 50? 100?) are worth and how important they are for the life of the voters.

This is actually a question that needs to be asked every time a crypto-system is installed that provides »mere« computational security against attacks, but of course its relevance in the field of online elections is not to be underestimated.

3.2.3 Personal Computer Security

One of the most important points raised in the discussions on whether remote internet voting is ready for a large-scale usage is the security of the typical personal computer.

To keep it short, the security of a typical personal computer is most of the time non-existent (default installation without any updates) or at least quite bad. Viruses or spyware which are targeted specifically at an upcoming online election pose a real threat to voters. Furthermore, phishing is a problem here as well: the voter has to make sure that he talks to the correct server – but without having to know too many details about public key infrastructures and related topics.

With new attacks like the memory cache timing side-channel attack on AES by Osvik, Shamir and Tromer ([OST05]), it is highly debatable whether secure voting is possible at all on a multi-user system. A special operating system which is only booted for the election might provide some help on the security issues but might still have other consequences – if a security problem is found within this OS, it is much easier to exploit because of the very homogenous environment.

Another alternative would be the combination of custom hard- and software, which might increase security as well but is not really cost-effective and has the same problem as the custom operating system.

Part II

Practical part

4 TUDvote – the proof of concept

> As a cryptography and computer security expert, I have never understood the current fuss about the open source software movement. In the cryptography world, we consider open source necessary for good security; we have for decades.
>
> *(Bruce Schneier, 1996)*

4.1 Background at TU Darmstadt

Technische Universität Darmstadt is a university of technology in south-Hesse, Germany. As the prototype deals with the specific situation at TU Darmstadt, some information on university elections at TU Darmstadt follows.

Every year in summer, university elections are held at TU Darmstadt. Every year, the student representatives in the *Universitätsversammlung* (the university assembly, which in turn elects the president of the university and the members of the senate) and the *Studierendenparlament* (student's parliament, which elects the members of the student union) are elected. Furthermore, committees within the departments, such as *Fachschaftsrat* and

Fachbereichsrat are elected as well. The former consists of lists of people, whereas in the latter election, one can normally elect candidates directly. Eligible voters are all students enrolled at TU Darmstadt, at the moment about 17.000. Normally, the turnout is quite low, but this has changed with the introduction of a law requiring more than 25% participation for the student union to be able to collect money from the students. Last year, turnout was more than 40%, which is quite some tallying work. Every other year, the employees and professors elect their representatives as well.

Furthermore, since last year, smart cards have been introduced for all students with a digital identity, which can be used to authenticate users for university applications. This smart card would be a possible authentication token for a real implementation of an online voting scheme.

4.2 Introduction to TUDvote

TUDvote is the name chosen for the prototype implementation of the Smith protocol. The implementation was done in Java and consists of a client, a server and a tallier. It is thus possible to actually try out the expected voting process from start till end, even though the software leaves out some vital parts of the program and is thus not actually secure for real-world use.

The software is missing the distribution of the private key to different entities and the zero-knowledge proofs that ensure that neither voter nor election authority can cheat. All the other parts of the protocol have been implemented though to show that such an implementation is indeed possible.

In the following, some of the design decisions will be presented, and after a package overview (which part of the source can be found where?), we

will have a more detailed look into the implementation. Information about how to install the software concludes this chapter.

4.3 Design decisions for the implementation

4.3.1 Java as programming language

As programming language for the proof of concept implementation, Java was chosen. This decision came mostly from the usability of Java within a web-browser and its high level of platform independence. As the final implementation of online voting for university elections should be as easy to use as possible, this is a reasonable step. Voters can then go to a special election website to which the connection is secured using HTTPS. There, the browser automatically downloads and starts the TUDvote applet which leads the voter through the voting process. This is much easier both for the user and the university than supporting an installation on a variety of different platforms.

Technically inclined users are of course encouraged to use different clients which implement the protocol to show that the security of the online voting process actually lies mostly in the protocol and is not depended on one special software implementation. The exact specification of the communication protocol as well as the release of a reference implementation should make the life of those easier who would like to implement an alternative client.

4.3.2 Implementation as open source software

The prototype has been released under the MIT license, an open source software license. To gain the trust of the potential users, a final implementation would be expected to be open source as well. This is actually the only way to guarantee that experts may look at the source and see what is going on. In a closed source voting software, information about who voted for whom could for example be sent out via side channels. This would be nearly impossible to notice for an observer who is only able to see the communication.

4.3.3 Unit testing using JUnit

To strengthen the robustness of the prototype, it was decided to use unit testing to assess the correctness of the individual parts.

4.3.4 Database access using HSQLDB

For the prototype, the Hypersonic SQL database (HSQLDB) engine was used. HSQLDB, which is used in well-known software products, such as JBoss, Mathematica and the OpenOffice.org 2.0 Office suite, is a lightweight database in Java, which offers many advanced features, a decent performance, and is very easy to install. Furthermore, it is available under a BSD-style open source license. More information on HSQLDB is available at [Hsq].

In the real software, a more powerful standalone database server will probably be used. It should be very easy to exchange the database because it is hidden by using a JDBC (Java Database Connectivity) driver as an abstraction layer.

4.3.5 Communication between client and server using XML-RPC

For the prototype, the communication channel between voter (client) and server (election authority) is implemented using XML-RPC. XML-RPC is a specification for remote procedure calling (RPC) over an HTTP (Hyper-Text Transfer Protocol) transport layer using XML as encoding. Basically, it offers a possibility to call remote methods from within various languages over a network. There are implementations of XML-RPC (for most of them both servers and clients are available) for various programming languages such as Perl, Python, C, C++, Java, PHP, Ruby, etc. running on many different platforms. More information and the specification can be found at the official XML-RPC website ([XML]).

Here, the XML-RPC implementation of the Apache Software Foundation was used. It offers a builtin webserver but can also be used within a servlet environment, which enables us to secure the communication channel using Transport Layer Security (TLS). A free servlet container that offers TLS is for example the well-known Tomcat server, which has also been released by the Apache Software Foundation. Information about how XML-RPC, Tomcat and TLS work together can be found in the server installation manual, cf. section 4.5.1.

4.3.6 Securing the connection using Transport Layer Security

For the communication between client and server, a communication channel that is protected against tampering with messages and that offers authentication of the server was needed. Therefore, TLS in combination with XML-RPC (cf. above) was implemented. TLS, the successor of Secure Sockets Layer (SSL), is an international standard specified by the Internet Engineering Task Force (IETF) in RFC 2246 (cf. [DA99]) which enables en-

cryption of application level protocols. It is used quite extensively on the internet, most prominently in securing websites using HTTPS, i.e., HTTP and TLS. Roughly speaking, this is the technology we use here: XML-RPC over HTTPS.

Certificate Authorities can be used together with TLS to make sure that the server is who it claims to be, so that the user talks to the real and not a faked election authority. The trust in the server could furthermore be strengthened by publishing hashes of its public key widely, for example on the polling cards that are sent out to the voters or on the posters that promote the election.

Note that HTTPS can be used with proxies and is most probably allowed when using firewalls, even very restrictive rulesets normally allow HTTPS, which is why it is often used for tunneling traffic as well. Furthermore, note that the security of the HTTPS channel used here is only necessary during the election time. If TLS is broken sometime between the elections, collected data packets can be decrypted, but the transmitted information is public anyhow, so this does not help an attacker.

4.4 The implementation

4.4.1 Package overview

The TUDvote implementation consists of three different packages, namely client and server packages »`de.tud.cdc.tudvote.client`« and »`de.tud.cdc.tudvote.server`« as well as a common package named »`de.tud.cdc.tudvote.common`«. The hierarchy `de.tud.cdc` is the one that is commonly used at the Cryptography and Computeralgebra group at the department of computer science at TU Darmstadt.

The client package corresponds to the software that runs on the voter's machine, the server package implements the election authority part, and the common package implements methods and represents data that are usable in both the client and the server. The XML-RPC connection classes are outside of this package hierarchy, but do most of the work on the server part as well as providing the XML-RPC server as a Java servlet.

In the following, a quick overview of the packages and their classes will be provided. The implementation details will be more focused on in the following sections, where what is happening during the different phases – initialization, voting, tallying – is explained in detail.

The de.tud.cdc.tudvote.client package

The »client« package consists of the following classes:

- TudVote
- Client
- UserInterface
  - TextUserInterface

The *TudVote* class is the main class, whose main method is started from the JAR file to run the TUDvote client program. It makes use of the JSAP (Java Simple Argument Parser) package to parse the command line options given by the voter. The *Client* class is instantiated in the main program and does all of the calculations and communications with the election authority (or rather the XMLRPC-server representing the election authority) that are needed to successfully submit a ballot. The *UserInterface* class is an abstract class that is used to keep all human computer interaction abstract enough to be able to implement different user interfaces. One such interface that is already implemented is the *TextUserInterface*, which asks the

user the relevant questions on the console. A possible addition would be a GraphicalUserInterface class, that solves the same tasks in a graphically more appealing way.

The de.tud.cdc.tudvote.server package

The »server« package consists of the following classes:

- Database
- DatabaseTest
- ServerInformation
- Tallier

The *Database* class is used for accessing the HSQLDB database (cf. section 4.3.4). The database is used to store both information about the voters (real name, student id, shared secret) as well as temporary information and the final ballots which are available on the public bulletin board. A JUnit test class is available as *DatabaseTest*, which tests whether the database works correctly. The *ServerInformation* class represents information that is needed by the server to do its work, for example methods that return its private key or its private signing key. The *Tallier* class is used to tally the final result of the election and consists of a main method that reads the relevant data from the database and computes the tally of the election.

The de.tud.cdc.tudvote.common package

The »common« package is by far the biggest, as many concepts and functions are needed both by the client and the server. It consists of the following classes:

- Ballot
- BallotPaper
- Candidate
- EncryptedBallot
- FinalBallot
- CommonInformation
- Group
  - ZModZP
- ElGamal
- ElGamalPrivateKey
- ElGamalPublicKey
- ElGamalSignature
- ZKPReEncryption
- ZKPValidity

and some corresponding JUnit test classes:

- BallotPaperTest
- ZModZPTest
- ElGamalTest

The *Ballot* class provides an abstraction of a ballot as it is commonly used on paper. Therefore, it consists of a *BallotPaper* class, which represents an empty piece of ballot paper, listing the *Candidate*s, and the corresponding votes. The *BallotPaper* class is used by the server to send a list of candidates to the client, who then creates a *Ballot* object that represents his choice. This ballot is then transformed into an *EncryptedBallot*, which is to be sent to the server. After all steps have been completed, the server has a *FinalBallot* containing all the information to be posted on the bulletin board.

The *CommonInformation* class is used to represent information that is needed by both the server and the client, such as the server's public keys and the server's public group element i, which is used during the encryption of the votes.

The abstract class *Group* provides an abstraction of a multiplicative group and their elements (implemented as an inner class *Group.GroupElement*). The subclass *ZModZP* implements this structure for a group of the form $\mathbb{Z}/p\mathbb{Z}$. Furthermore, this group has an additive structure, which is used for example with ElGamal signing. This is why this part, which exceeds the specification of the *Group* class, is implemented here as well. The *ZModZPTest* class provides JUnit tests, which test the correctness of the implementation. This is done for example by checking whether $a \cdot a^{-1} = 1$ holds for a random group element a and various other checks which largely mirror the axioms for groups.

The *ElGamal* class is a static class which implements the various tasks of the ElGamal algorithm, i.e., en- and decryption, signing and signature checking, as well as re-encryption. The datatypes used for these tasks are implemented in the *ElGamalPrivateKey*, *ElGamalPublicKey* and *ElGamalSignature*, which provide data structures for private keys, public keys and signatures. The corresponding JUnit class *ElGamalTest* implements unit tests for this class, for example checking whether the signing function returns a correct

signature which can be verified, or whether encrypted messages can be decrypted using the respective private key.

At the moment, the classes *ZKPReEncryption* and *ZKPValidity* are more or less empty classes, which provide a framework to integrate the zero knowledge proofs of correct re-encryption and of validity of votes into a later version of the software.

Classes outside of package hierarchy

The *ElectronicVotingXMLRPC* and *XMLRPCClass* classes are two classes that live outside of the regular package hierarchy, as they are used in the XML-RPC server. ElectronicVotingXMLRPC is a Java servlet that instantiates an XMLRPCClass and passes on incoming XML-RPC requests to the class. So if, for example, the XML-RPC function »getCandidateList()« is requested by a client, the corresponding method of the XMLRPCClass is called to do the calculation. This entails that most of the actual work on the server side is implemented here.

4.4.2 The voting phase

In figure 2.3, the voting protocol communication is described. To implement this communication in software, similar figures were created to provide an overview of which techniques and algorithms are needed and which methods are used for each step. To provide a reminder of what is done when (without skipping back to figure 2.3 on page 25), the original, higher-level description of the corresponding step is shown at the top of the figure, whereas the technical implementation details are shown as »sub-steps«. The arrows in the pictures do not necessarily depict communication activity, but rather which side is involved in the computation. We start at step number four, as the initialization steps which should take place

beforehand are unneccessary in the prototype version – they only deal with distributing the private election authority key. In the following, we will show what happens when the client is started, i.e., the main() method of the de.tud.cdc.tudvote.client.TudVote class is called.

Step 4: send ElGamal public key and pre-shared secret

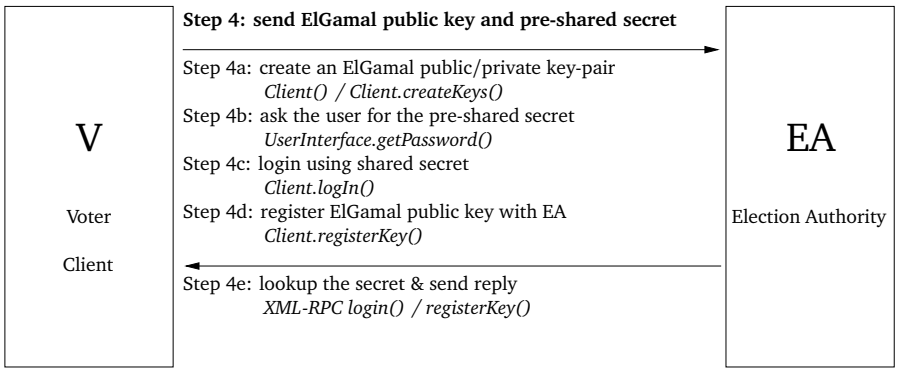

Figure 4.1: Voting phase implementation, step 4

Before actually sending the public key, the client logs into the server to receive an authentication cookie, which will be used as a session identifier in the future. The client thus calls the XML-RPC function »login()« with the username and a SHA256 hash of the password (which is assumed to be random enough so that specific password techniques such as salting are not needed). The server reacts by checking whether the user is in the database table »voters« and the password matches. It creates a random 160bit number, which is returned as a hex-string to the user. This authentication cookie is saved in the »auth« table in the database and used later on to authenticate the user and match him to an existing session.

Afterwards, the client creates a new random ElGamal public/private keypair, of which the public key is sent to the server using the XML-RPC function »registerKey()«. The key uses the same group as the server, which is the 2048 bit $\mathbb{Z}/p\mathbb{Z}$ group mentioned in RFC 3526 – More Modular Exponential (MODP) Diffie-Hellman groups for Internet Key Exchange (IKE). The prime used here is $2^{2048} - 2^{1984} - 1 + 2^{64} \cdot (\lfloor 2^{1918} \pi \rfloor + 124476)$. Using this prime has the distinct advantage that the generation of a random prime is unnecessary and that this prime is considered secure, as it is used widely in the Internet Key Exchange protocol, a protocol to exchange keys for IPsec communication. The server stores this key in the »pubkeys« tables of the database and returns a boolean to the client to indicate whether the registration was successful.

Step 5: send candidate list

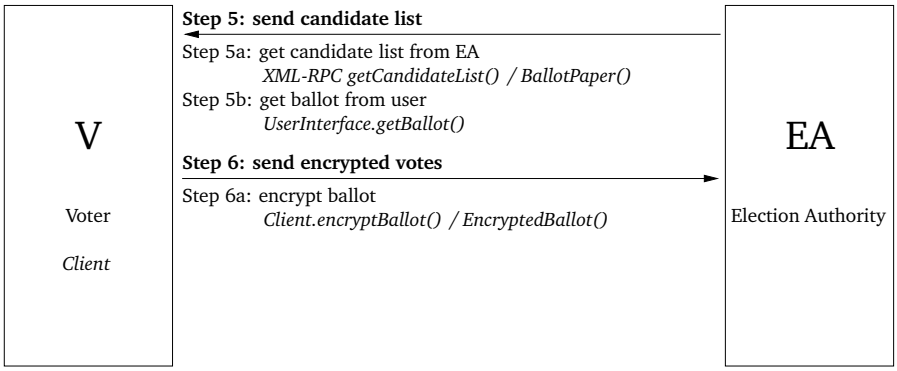

Figure 4.2: Voting phase implementation, step 5 & 6

To get the candidate list from the server, the client calls the XML-RPC function »getCandidateList()« with his authentication cookie and the identification number of the committee he wants to vote for. The server checks

whether the user is authenticated by looking up his cookie in the corresponding database table and opens a text file corresponding to the committee number to read in the information about the candidates. A typical text file for a TUD university election might look like this:

```
1;Studierendenparlament;7;1
1;FACHWERK;0,1
2;Jusos und Unabhängige;0,1
3;Liberale Studenten Darmstadt;0,1
4;RCDS & Unabhängige;0,1
5;Liste Odenwald;0,1
6;Bund internationalistischer Studierender (BISS);0,1
7;Die Grünen;0,1
```

The first line consists of the committee id, the committee name, the number of candidates and the maximum number of votes. All other lines provide information about the candidates: a number to identify them, a name and a list of possible votes. Here, one is only able to vote either 0 or 1, resp. »no« or »yes«, but in general it might be possible to have more than one vote for each candidate. The server parses this list and creates a new BallotPaper object from it, which it sends as a string to the client. The client on the other hand again creates a BallotPaper object which is used to get the ballot from the user – either from the command line information provided by the user (which is actually only useful for testing or debugging purposes) or from the UserInterface object via the getBallot() method. This leaves us with a Ballot object that can be used later on.

Step 6: send encrypted votes M_j

Before the encrypted votes can be sent, the Ballot has to be encrypted first, this is done using the encryptBallot() method in the Client class, which just

calls the corresponding constructor of the EncrypedBallot class. Here, the actual encryption takes place, i.e., for all votes in the ballot, i^{v_j} is calculated and encrypted using the election authority's public key and saved in an array of ElGamalEncryptedMessages.

Step 7: send re-encrypted votes M'_j

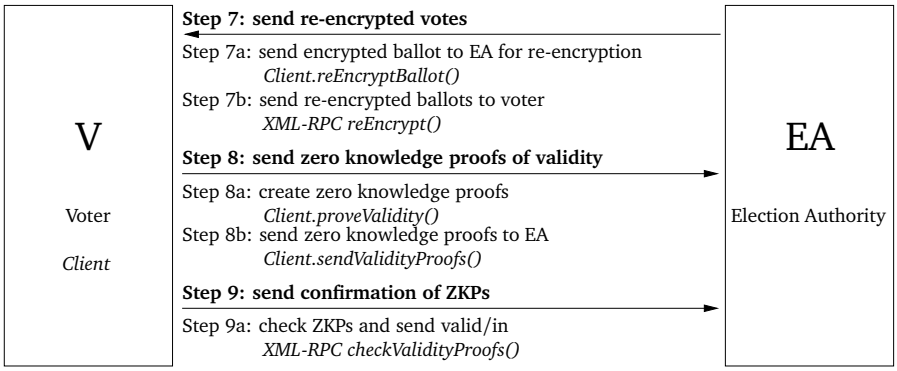

Figure 4.3: Voting phase implementation, step 7–9

The actual sending of the encrypted messages is now part of step 7: the main program calls the Client's reEncryptBallot() method, in which the client sends every single vote to the server to be re-encrypted – using the XML-RPC function »reEncrypt()«. This takes the authentication cookie, the committee and candidate ids and a string representation of the encrypted vote as an argument. The server re-encrypts it by using the corresponding function from the ElGamal class (which just multiplies the vote by an encryption of one) and saves the student id, the committee and candidate id, as well as the re-encrypted vote into the database in the reencryptions table. This table is then later on used to verify that the re-encrypted vote

submitted by the voter has actually been re-encrypted by the election authority. The re-encrypted vote is then returned to the voter as a string, from which he can create ElGamalEncryptedMessages and, combining all of them, another EncryptedBallot object.

Step 8: send zero knowledge proofs of validity

The creation of the zero knowledge proofs of validity is to be done in the proveValidity() method of the Client object. This method creates a new ZKPValidity object from an encrypted ballot, by calling the corresponding constructor of the ZKPValidity class. Currently, this creates an empty proof without any meaning to it, which will always be recognized as valid.

Step 9: send confirmation of zero knowledge proofs – valid/invalid

The ZKPValidity object is then sent to the server in String form accompanied by a signature on the String in the Client method sendValidityProofs(). This is done by calling the XML-RPC function »checkValidityProofs«, which, on the server side, verifies the signature and the validity of the proof and saves them away in the zkpsvalidity table in the database. It returns a boolean to the user indicating whether both the zero knowledge proof and the signature were correct.

Step 10: send designated verifier zero knowledge proofs for re-encryption

As the designated verifier zero knowledge proof of correct re-encryption has not been implemented yet, there is currently only a stub in the Client

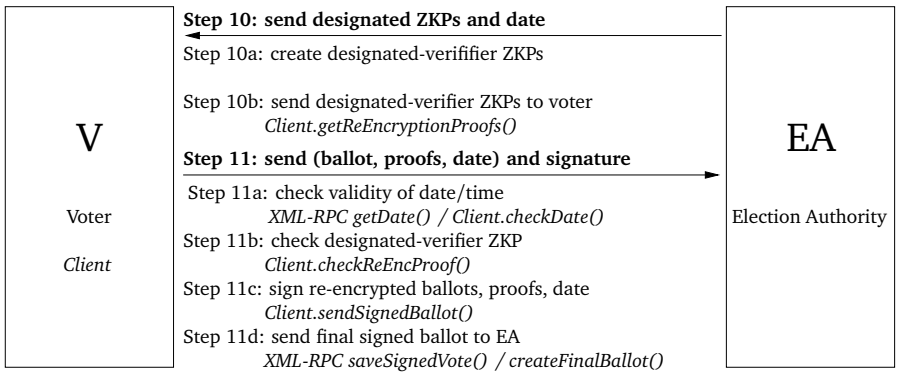

Figure 4.4: Voting phase implementation, step 10 & 11

class called getReEncryptionProofs(), which at the moment does not communicate with the server at all. In later versions, this should get the proof from the server and they should be verified to be sure they are correct. Another stub here is the checkReEncProof() method, which is supposed to do the actual checking of the validity.

The working code in this part is about the date that is contained in the information posted on the public bulletin board by the server: getDateFromServer is called in the Client class, which calls the XML-RPC function »getDate()« on the server side, which looks up the current date, saves it together with the username and committee id in the dates table of the database and returns it to the user. On the client side, this date is compared against the local date using checkDate() in the client class. If it differs more than a maximum difference (currently set to 60 seconds), it asks the user whether he wants to accept the date anyway using the UserInterface method askUserForDateCorrectness().

Step 11 & 12: submit final signed ballot

To submit the final signed ballot, the Client method sendSignedBallot() is called. For every candidate, it creates a formatted string containing the username, the committee id, the candidate id, the re-encrypted vote for that candidate and the server publication date. This string is then signed and both the string and the signature are sent to the server via the XML-RPC function »saveSignedVote()«. The server then checks the signature and whether the information given to him is correct, i.e., whether the username, the ids, the vote and the date match the information that he has saved in his database during the course of the protocol. If he accepts the vote, he saves the relevant information in the database table finalvotes and returns a boolean indicating his acceptance to the user.

Step 13 & 14: post final ballot to bulletin board

Afterwards, the client calls the XML-RPC function »createFinalBallot()« to state that he is done submitting votes and would like to submit the ballot to the bulletin board. The server then pulls the relevant votes from his finalvotes table and appends them and the corresponding zero knowledge proofs to each other, seperated using the »|« character. It creates a signature and saves the final ballot and the corresponding signature in the finalballots table of the database and returns a boolean indicating whether saving them was successfull. In the prototype, the client is unable to get back the final ballot from the server (as it is used for tallying only), but in a real implementation, this should be offered, as the signed final ballot is a proof for the user that the server accepted his ballot. This will be done using the getFinalBallot() method in the Client class and the outputFinalBallot() method in the UserInterface class. This would also be the place to output them encoded as a barcode, for example.

4.4.3 The tallying phase

After all votes have been accepted by the server, the tallying phase can start. The prototype tallier is implemented in the *Tallier* class in the server package. It gets all submitted votes from the finalballots table of the database and multiplies the tally for each candidate with the corresponding encrypted vote. Afterwards, it decrypts the tallies to receive i^{T_j}. It computes the discrete logarithm by test-multiplying i with itself until it reaches the correct value. Note that this is fast enough to take just a few seconds. It outputs the tally for each candidate and then quits. At the moment, this is at the moment a very minimalist tallier. A real implementation should check all of the security features that are embedded in the data available on the database as well – most prominently the signatures and zero knowledge proofs.

4.4.4 System testing strategy

To test the correctness of the system as a whole, a system test was implemented. Therefore, the command line client was used which was called from a Perl script to vote pseudo-randomly. The result was once tallied using the input from the Perl script and once at the end of the test election by the talliers to verify that both results match. A typical test session looks like this:

```
[klink@calendula]:~/src/classes$ time ~/test_voting.pl 15
012345678901234
ballot[0] = 1
ballot[1] = 4
ballot[2] = 2
ballot[3] = 2
ballot[4] = 3
ballot[5] = 3
ballot[6] = 0
```

```
real    17m16.230s
user    4m16.148s
sys     0m3.266s

[klink@calendula]:~/src/classes$ time java \
de.tud.cdc.tudvote.server.Tallier
i^(Tally) for candidate 1:
174682681729795286751730342719251931379921941882022398668...
DL is: 1
i^(Tally) for candidate 2:
320169128940412095918462545388614089701070825779880998451...
DL is: 4
i^(Tally) for candidate 3:
420466337220673914548727884275951764846044714160203629053...
DL is: 2
i^(Tally) for candidate 4:
420466337220673914548727884275951764846044714160203629053...
DL is: 2
i^(Tally) for candidate 5:
306744212980309294659730544631682370647917101572560732492...
DL is: 3
i^(Tally) for candidate 6:
306744212980309294659730544631682370647917101572560732492...
DL is: 3
i^(Tally) for candidate 7:
1
DL is: 0

real    0m7.309s
user    0m7.026s
sys     0m0.112s
```

The perl script is run with an option indicating how many different random ballots it should submit. One can see that using the standard Unix `time(1)` command, one can measure the speed as well – and that the current implementation is pretty slow. More on why and how to avoid this in a real implementation can be found in the »Conclusions & Further work« chapter.

The tally is then computed using the Tallier program from the ballots saved in the database. We can see that result is actually the one that we expected from posting the corresponding ballots and that the tallying for our small example can be done within a few seconds.

4.5 Installation of the server software

4.5.1 Installation of the servlet container

For the prototype, Apache Tomcat, the servlet container implementation of the Apache Software Foundation was used. Tomcat offers Transport Layer Security (TLS) support and is offered under a very liberal open source license and was thus chosen to be the servlet container of choice for the project.

Installation documentation and packages are provided at Tomcat's website ([Tom]). For the prototype, Tomcat 5.0 was used but later versions should work as well. Alternatively, other servlet containers with TLS support may be used but those may require different installation procedures or even a change of the client code.

Once Tomcat is installed (on a Debian-based system, this is as easy as entering »`apt-get install tomcat5`«), it needs to be configured properly to support TLS. To be able to use TLS, a private key for the server needs to be produced to create a certificate. The certificate then needs to be exported as a Certificate Signing Request, which is signed by the certificate authority that is responsible for it. In the prototype implementation, this is done using a certificate authority responsible for the electronic voting, but this might differ based on policies.

Here is an example of how to create the certificates and the corresponding keystore works:

```
calendula:/usr/share/tomcat5# keytool -genkey -alias tomcat \
                                      -keyalg RSA \
                                      -keystore .keystore
Enter keystore password:  changeit
What is your first and last name?
  [Unknown]:  evote.tu-darmstadt.de
What is the name of your organizational unit?
  [Unknown]:  Electronic Voting
What is the name of your organization?
  [Unknown]:  Technische Universität Darmstadt
What is the name of your City or Locality?
  [Unknown]:  Darmstadt
What is the name of your State or Province?
  [Unknown]:  Hessen
What is the two-letter country code for this unit?
  [Unknown]:  DE
Is CN=evote.tu-darmstadt.de, OU=Electronic Voting,
  O=Technische Universität Darmstadt, L=Darmstadt,
  ST=Hessen, C=DE correct?
  [no]:  yes

Enter key password for <tomcat>
        (RETURN if same as keystore password):
```

To create a Certificate Signing Request, the following command can be used:

```
calendula:/usr/share/tomcat5# keytool -certreq -keyalg RSA \
                                      -alias tomcat \
                                      -file certreq.csr \
                                      -keystore .keystore
Enter keystore password:  changeit
```

The file `certreq.csr` can then be sent to the corresponding CA which signs the certificate and sends it back. For the server to work properly, both the CA certificate as well as the signed certificate need to be imported into the keystore using commands similar to the following:

```
calendula:/usr/share/tomcat5# keytool -import -alias root \
                                      -keystore .keystore \
                                      -trustcacerts \
                                      -file cacert.pem
calendula:/usr/share/tomcat5# keytool -import -alias tomcat \
                                      -keystore .keystore \
                                      -trustcacerts \
                                      -file signedcert.pem
```

This concludes the process of creating the necessary certificates for running the Tomcat server, but it still needs to be configured to actually use TLS. Therefore, a TLS Connector needs to be added in the `server.xml` configuration file, which should look similar to this:

```
<Connector className="org.apache.coyote.tomcat5.CoyoteConnector"
    port="443" minProcessors="5" maxProcessors="75"
    enableLookups="true" disableUploadTimeout="true"
    acceptCount="100" debug="0" scheme="https" secure="true"
    clientAuth="false" sslProtocol="TLS"/>
```

One should make sure to delete all containers that listen on port not using encryption to avoid leaving open unneeded ports. If unsure, a port scanner can be used to check whether Tomcat listens on any other ports as well. Note that running on port 443 requires Tomcat to run as root under Unix. A better solution might be to run it on a non-priviliged port and then use software such as rinetd to forward port 443 to the non-priviliged port.

4.5.2 Installation of the server software

To deploy the software, one has to create a web application in the servlet container, for example by creating a directory below `${catalina.home}/`

webapps/. In the prototype case, this directory is called tudvote. To configure this web application, a file named web.xml in the subdirectory WEB-INF has to be created. A minimal working file looks like this:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE web-app
    PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
    "http://java.sun.com/dtd/web-app_2_3.dtd">

<web-app>
    <servlet>
        <servlet-name>ElectronicVotingXMLRPC</servlet-name>
        <servlet-class>ElectronicVotingXMLRPC</servlet-class>
    </servlet>

    <servlet-mapping>
        <servlet-name>ElectronicVotingXMLRPC</servlet-name>
        <url-pattern>/ElectronicVotingXMLRPC</url-pattern>
    </servlet-mapping>
</web-app>
```

Afterwards, the server can be configured by editing the ServerInformation.java file, where the location of the candidate information, the location of the database (which can be created by copying the appropriate files from the templates directory) as well as the private keys can be defined. To allow the server access to the database, permissions have to be added to the Tomcat policy files, for example similar to this template:

```
grant codeBase "file:${catalina.home}/webapps/tudvote/WEB-INF/-" {
 permission java.io.FilePermission
   "/home/klink/stud/thesis/db","read";
 permission java.io.FilePermission
   "/home/klink/stud/thesis/db/-","read,write,delete";
 permission java.io.FilePermission
   "/home/klink/stud/thesis/candidates","read";
 permission java.io.FilePermission
```

```
    "/home/klink/stud/thesis/candidates/-","read";
  permission java.security.SecurityPermission
    "insertProvider.FlexiCore";
  permission java.security.SecurityPermission
    "putProviderProperty.FlexiCore";
};
```

Copying the compiled ElectronicVotingXMLRPC.class, XMLRPCClass.class files and the hierarchies `de/tud/cdc/tudvote/common` and `de/tud/cdc/tudvote/server` to the classes subdirectory of the WEB-INF directory and copying the JAR files for FlexiProvider, HSQLDB, Commons Codec and XML-RPC to the lib subdirectory, the installation of the XML-RPC server is ready to work.

4.6 Installation of the client

Before packaging the client into a JAR file, it has to be configured for the specific installation of the server. The relevant changes can be done in ClientInformation.java. First of all, one needs to specify the location of a keystore file which contains the certificate of the Certificate Authority. It can be created using »keytool -keystore cacert_local -import -trustcacerts -file cacert.pem«, where cacert.pem is the PEM file containg the CA's key. Furthermore, the URL of the server has to be adapted to the settings. Once this customization is done, the client can be compiled an packaged into a JAR file. Therefore, the hierarchy `de/tud/cdc/tudvote/client` and `de/tud/cdc/tudvote/common` as well as the content of all relevant JAR files (FlexiProvider, JSAP, XML-RPC, Commons Codec) have to be packed into a JAR file. This JAR file should contain a manifest file which specifies de.tud.cdc.tudvote.client.TudVote as the main class.

Having done this, the voter can easily start the client by running »java -jar tudvote.jar«.

5 Practical security considerations

Machines take me by surprise with great frequency.

(Alan Turing)

In this chapter, we will look at the practical security of an electronic voting implementation. Therefore, we look at different attack scenarios. One of them would be to compromise the indivudal votes, another one to influence the election. Yet another one would be making the election impossible at all. As the current implementation easily allows some of the attacks, we assume that we have an implementation that fixes these shortcomings, i.e., implements zero knowledge proofs and distributed keyholders.

5.1 Compromising the individual votes

As the votes are only available in an unencrypted form on the client computer (at the time of entry), the easiest and most promising attack would be to compromise the voter's machine. Specialized viruses and malware could be written to send out the vote in an unencrypted form to an adversary. Considering the security of today's personal computers, this is quite a realistic threat if the interest in it is high enough. A compromised PC can of course be also used to show the user the voting dialogue and let him decide for a specific ballot, but vote in a different way afterwards. This is

a special case of the more general »display problem«: with cryptography, the user never knows that what he sees is what is actually computed and encrypted. So this is the worst-case scenario for the individual voter, but unluckily it is not very unlikely.

Compromising the votes at the network level or at the server side would not help the adversary, as they are only available in encrypted form at that stage and can only be decrypted by the keyholders together. A compromise of a large enough subset of keys of the keyholders would then be required, which is (given a decent implementation of the sharing and tallying software) far more unlikely than compromising a voter's machine. Of course, compromising the keyholders' keys is *the* single worst case scenario for the election system in total, as this would enable an adversary to decrypt any ballot that has been posted.

5.2 Influencing the election

Depending on your viewpoint, influence on an election may be seen as an even worse scenario for the election system, especially if it goes unnoticed. To influence the election, at least one of the following subgoals have to be achieved:

- Change the vote of a voter
- Delete the vote of a voter
- Vote disguised as a voter who has not voted (yet)

5.2.1 Changing the vote of a voter

Changing the vote of a voter could either be done at the client or at the server side. The change is less likely to be detected at the client side, espe-

cially if the voter believes he has sent a correct ballot himself (see above). At the server side, the impact would be much higher, but it is more difficult to achieve as well, as the vote has to be signed with a valid key which the voter submits at the start of the protocol. So this would require either changing the key, which in turn would require a man-in-the-middle position for the adversary, or access to the corresponding private key, which would require access to the voter's computer. So the natural position would again be at the client's computer, where the vote could be easily changed without the server noticing anything about it, as the client talks the correct protocol.

5.2.2 Deleting the vote of a voter

Deleting a voter's ballot from the database would take place at the server side. There is a possibility that an attacker might gain access to the database, either by having (physical or remote) access to the database server, or by using SQL-injections (a code review should possibly show whether there are any left). This way it would be possible to destroy only some or all of the votes. The latter would of course be noticed, but there is no real way to fix it. Deletion of only a few votes is much more critical. This might go unnoticed (even though the voter's whose votes are deleted could notice it on the public bulletin board and have the receipt by the election authority to prove that there vote has been accepted) and influence the election in a certain way (even though the attacker has to have statistical evidence as to which group votes how to influence the election in a particular way, as we assume he can not decrypt it). A solution against this might be to replicate the database to a medium which is append-only and thus not erasable.

5.2.3 Voting for someone else

To vote for someone else, who has not voted yet or is unlikely to vote (electronically) would require his authentication information, thus physical access to either his pre-shared secret on his *Wahlbenachrichtigung* or access to his chipcard and PIN. If those are not available, the election authority will plainly refuse to communicate with an impostor. In the case of pre-shared secrets, the election authority of course has to have access to them and is thus in a position to become an impostor itself. An organizational solution against this would be to seperate the authentication to a different entity and let the election authority only ask whether the user is authenticated (and log the corresponding response, so that in a case of a dispute, there is information that can prove something). This particular problem is an advantage for the chipcard solution, where the EA would have no access at all to the authentication information. On the other hand, this also introduces another party that has to be trusted to work correctly – the certificate authority that signs the certificates for the public keys on the smart cards. All in all, this is a possible attack scenario, but it seems much less likely then deletion or change of vote.

5.3 Making an election impossible

Another goal that an attacker might have is to disable the possibility of remote voting at all, thus limiting the possibility to participate in the election for voters and discrediting the electronic voting process. With the current prototype implementation, a single election authority is used to accept all the votes. Of course the word »single« in the last sentence already points to the fact that this is also a single point of failure. Luckily, the protocol theoretically allows for more than one election authority that can then be clustered or for example be addressed using round-robin DNS. This provides *some* level of protection against a denial of service attack on the server, but

it might not stop an adversary with enough power or money (note that botnets that do DDoS (distributed denial of service) attacks can be bought on the »free market« today). This seems unlikely for a university election (who could always have a policy to restrict voting to its internal network anyways), but has to be remembered if one thinks of higher-profile elections.

6 Conclusions & further work

As an overall conclusion of this thesis, we can see that an implementation of Smith's cryptographic voting scheme is possible, and that the theoretical paper can be transformed into a usable prototype. Even though, at the moment the speed is far from optimal, many improvements in this area are possible and it should be no problem to have a software that takes just a few seconds on a modern machine. One such point would be using groups of elliptic curves over finite fields instead of just $\mathbb{Z}/p\mathbb{Z}$. This should offer both speed and size benefits without losing security and would be easy to implement as a subclass of the current Group class. Regardless of that, for a real system, the representation of the ballots and their transfer over the network would have to be improved. For now, they are just represented as strings of their decimal representation, which is not exactly space-efficient. A common representation that could be realized by different clients would for example be possible by using ASN.1 (Abstract Syntax Notation One), which is a standard notation used for representing data.

But the first work that should be concentrated on is the implementation of the missing features. As already noted, these are the distribution of the secret key among different entities and the zero knowledge proofs. Sharing the key between different entities should be the easiest to implement, it is more or less pretty straight-forward, but one has to think of and implement a communication protocol to communicate between the keyholders. Note that the protocol requires a zero knowledge proof in this area as well – during the tallying phase, the keyholders have to prove that they use the same sub-key that they have used in the beginning. Zero knowledge proofs are the second big missing part, and they seem to be more difficult to implement. They are very well understood theoretically, but there seems to

be almost no implementation of them in a generic form. The framework for implementation is available in the code, but it would need to be filled with life.

Having done this, there should be a working implementation of the protocol available. This leaves us at the question whether this implementation is secure, which could only be solved by a massive code review or even the more difficult task of trying to prove (parts of) the code correct. This is a step that is absolutely required to gain trust in the system. At this point, a refactoring of the source code may be a good idea to encapsulate parts of the programs from each other as much as possible. Along with this, more logging and auditing capabilities should be included in the source.

Afterwards, one could think about adding new features to the system. One that springs to mind is the production of paper receipts, which should be pretty easy to implement. For instance, PDF417, a two dimensional barcode technology, allows for roughly 2kB of data per barcode. When using $\mathbb{Z}/p\mathbb{Z}$, the basic blocks are 2048bit long numbers, which means that for every ballot, a few barcodes would be needed. With the use of elliptic curves, these numbers become much smaller, and a representation of a ballot in one barcode seems possible. That means that a receipt for a whole election could be provided for the voter on a standard A4 paper. Another new feature would be the possibility to use the TUD smart card for authentication. This should be pretty easy to implement, and it should be discussed whether this should be the only way of authentication for the election.

All along the further work on the implementation, there should be a public discussion of the benefits and drawbacks of cryptographic voting in general, and for university elections at TU Darmstadt in particular. Based on this discussion, a decision has to be taken on whether to transform the prototype to a real-world implementation for a legally binding election.

Bibliography

[Acq04] Alessandro Acquisti. Receipt-free homomorphic elections and write-in ballots. Technical Report 2004/105, International Association for Cryptologic Research, May 2004.

[AR05] President of the Republic of Estonia Arnold Rüütel. The President of the Republic refused to promulgate an act concerning the possibility of e-voting at the local government council elections. Press release. `http://www.president.ee/en/duties/press_releases.php?gid=63360` (last access: March 12th, 2006), May 2005.

[Ben96] Josh Daniel Cohen Benaloh. *Verifiable Secret-Ballot Elections*. PhD thesis, Yale University, December 1996.

[BFP+01] O. Baudron, P.-A. Fouque, D. Pointcheval, G. Poupard, and J. Stern. Practical multi-candidate election system. In N. Shavit, editor, *Proceedings of the 20th ACM Symposium on Principles of Distributed Computing*, pages 274–283. ACM Press, New York, August 2001.

[BG02] Dan Boneh and Philippe Golle. Almost entirely correct mixing with applications to voting. In Vijay Atluri, editor, *CCCS '02: Proceedings of the 9th ACM conference on Computer and communications security*, pages 68–77. ACM Press, 2002.

[Buc04] Johannes Buchmann. *Introduction to Cryptography*. Springer, second edition, 2004.

[CC97] Lorrie Faith Cranor and Ron K. Cytron. Sensus: A security-conscious electronic polling system for the internet. In *Proceedings of the Hawaii International Conference on System Sciences*, 1997.

[CFSY95] Ronald Cramer, Matthew Franklin, Berry Schoenmakers, and Moti Yung. Multi-authority secret-ballot elections with linear work. Technical Report CS-R9571, Centrum voor Wiskunde en Informatica, 1995.

[CGS97] Ronald Cramer, Rosario Gennaro, and Berry Schoenmakers. A secure and optimally efficient multi-authority election scheme. In Walter Fumy, editor, *EUROCRYPT '97*, volume 1233 of *Lecture Notes in Computer Science*, pages 103–118. Springer-Verlag Berlin Heidelberg, 1997.

[Cha81] David Chaum. Untraceable electronic mail, return addresses, and digital pseudonyms. *Communications of the ACM*, 24(2):84–88, February 1981.

[Cha82] David Chaum. Blind signatures for untraceable payments. In *Proceedings of Crypto 82*, volume Advances in Cryptology, pages 199–203. Plenum Press, New York, 1982.

[DA99] T. Dierks and C. Allen. The TLS Protocol Version 1.0. RFC 2246 (Proposed Standard), January 1999. Updated by RFC 3546.

[DGS03] 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.

[FOO92] Atsushi Fujioka, Tatsuaki Okamoto, and Kazuo Ohta. A practical secret voting scheme for large scale elections. In Jeniffer Seberry and Yuliang Zheng, editors, *Advances in Cryptology – AUSCRYPT '92*, pages 244–251. Springer-Verlag, 1992.

[Hsq] The hsqldb Development Group. *HSQL database engine – Lightweight 100% Java SQL Database Engine*, 2005. `http://www.hsqldb.org` (last access: October 22nd, 2005).

[Int] Forschungsgruppe Internetwahlen. Zweiter Zwischenbericht zum Projekt »Strategische Initiative: Wahlen im Internet« nach Abschluss der Wahlen zum Studierendenparlament der UOS am 2. Februar 2000. `http://www.wahlkreis300.net/fgiw/uploader/data/stupa.pdf` (last access: March 12th, 2006).

[JJ99] Markus Jakobsson and Ari Juels. Millimix: Mixing in small batches. Technical Report 99-33, DIMACS, June 1999.

[JL97] Wen-Shenq Juang and Chin-Laung Lei. A secure and practical electronic voting scheme for real world environments. *TIEICE: IEICE Transactions on Communications/Electronics/Information and Systems*, E80-A(1):64–71, 1997.

[JRSW04] David Jefferson, Aviel D. Rubin, Barbara Simons, and David Wagner. Analyzing internet voting security. *Communications of the ACM*, 47(10):59–64, October 2004.

[KSRW04] Tadayoshi Kohno, Adam Stubblefield, Aviel D. Rubin, and Dan S. Wallach. Analysis of an electronic voting system. In *IEEE Symposium on Security and Privacy*, 2004.

[KY04] Aggelos Kiayias and Moti Yung. The vector-ballot e-voting approach. In *8th International Conference on Financial Cryptography 2004*, volume 3110 of *Lecture Notes in Computer Science*, pages 72–89. Springer-Verlag, 2004.

[Lee99] Jong-Hyeon Lee. The big brother ballot. *Operating Systems Review*, 33(3):19–25, 1999.

[LK00] Byoungcheon Lee and Kwangjo Kim. Receipt-free electronic voting through collaboration of voter and honest verifier. In *Proceeding of JW-ISC2000*, pages 101–108, January 2000.

[LK02] Byoungcheon Lee and Kwangjo Kim. Receipt-free electronic voting scheme with a tamper-resistant randomizer. In *Proceedings of ICISC 2002*, volume 2587 of *Lecture Notes in Computer Science*, pages 389–406. Springer-Verlag, 2002.

[MH96] Markus Michels and Patrick Horster. Some remarks on a receipt-free and universally verifiable mix-type voting scheme. In *ASIACRYPT: Advances in Cryptology – ASIACRYPT: International Conference on the Theory and Application of Cryptology*, Lecture Notes in Computer Science. Springer-Verlag, 1996.

[Nef04] C. Andrew Neff. Verifiable Mixing (Shuffling) of ElGamal Pairs. `http://www.votehere.net/vhti/documentation/egshuf.pdf` (last access: March 12th, 2006), April 2004.

[Oka97] Tatsuaki Okamoto. Receipt-free electronic voting schemes for large scale elections. In *Security Protocols Workshop*, pages 25–35, 1997.

[OST05] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and Countermeasures: the Case of AES. Cryptology ePrint Archive, Report 2005/271, 2005. `http://eprint.iacr.org/2005/271.pdf` (last access: March 12th, 2006).

[RRN01] Indrajit Ray, Indrakshi Ray, and Natarajan Narasimhamurthi. An anonymous electronic voting protocol for voting over the internet. In *Proceedings of the Third International Workshop on Advanced Issues of E-Commerce and Web-based Information Systems*, 2001.

[Rub] Avi Rubin. Security considerations for remote electronic voting over the internet.

[Sch99] Berry Schoenmakers. A simple publicly verifiable secret sharing scheme and its applications to electronic voting. In Michael Wiener, editor, *Advances in Cryptology—CRYPTO' 99*, volume

1666 of *Lecture Notes in Computer Science*, pages 148–164. Springer-Verlag Berlin-Heidelberg, 1999.

[Smi05] Warren D. Smith. Cryptography meets voting. Chapter from the upcoming book »How Mathematics can Improve Democracy«, March 2005.

[Tom] The Apache Software Foundation. *Apache Tomcat*, 2005. `http://tomcat.apache.org` (last access: October 21st, 2005).

[Tru04] Tom Trumpbour. Voting machine technology. In *Proceedings of the 21st Chaos Communication Congress*, pages 253–257, December 2004.

[Wik04] Douglas Wikström. A universally composable mix-net. In Moni Naor, editor, *Theory of Cryptography*, pages 317–335. Springer, February 2004.

[XML] Scripting News, Inc. *XML-RPC Home Page*, 2005. `http://www.xmlrpc.com` (last access: October 22nd, 2005).

Contents of the enclosed CD-ROM

This thesis comes with a CD-ROM which includes the source code for the prototype implementation. The source is also downloadable at `http://www.klink.name/diplom/`.

The layout of the CD-ROM is as following:

- tudvote_source.jar – the source of the system as a JAR-file.
- source/ – this directory contains the source of the system as well as a build.xml ant file and the required third-party libraries.
- templates/ – this directory contains templates, such as an empty database file and an example candidate configuration file.
- helper/ – this directory contains helper scripts, such as a SQL script to clear the database and the perl script used for testing the voting.
- thesis/ – this directory contains this thesis as Postscript and Portable Document Format files.