Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Distributed Computing

Anonymous and Transparent E-Voting System

Bachelor Thesis

Lucien Schaller

slucien@student.ethz.ch

Distributed Computing Group
Computer Engineering and Networks Laboratory
ETH Zürich

Supervisors:
Conrad Burchert
Prof. Dr. Roger Wattenhofer

March 5, 2018

Acknowledgements

I thank Conrad Burchert, who gave me great advice on many questions throughout this thesis, and who was always passionate about the topics of our meetings.

Abstract

In this thesis we design an E-Voting System. The aim was to design a system in which the voter's anonymity is preserved, meaning nobody can trace the vote to the person which submitted it. Everyone should be able to reproduce the voting process which makes it publicly verifiable. Based on the design, we implemented a server back-end and a front-end desktop application, to hold and verify a voting.

Contents

Chapter 1

Introduction

People are reluctant to trust current e-voting system implementations, because of the impacts on voter privacy and transparency. There have been several experiments by different countries to apply such a system, but detecting fraud has been difficult.
An electronic voting system has several requirements that should be fulfilled. Some of those are the following:

- **Anonymity**: Nobody should be able to find out what a certain voter voted for
- **Authenticity**: The system has to authenticate every user and make sure only eligible people can vote
- **Individual verifiability**: Every voter can check that its vote has been counted correctly
- **Universal verifiability**: Every voter can check the correctness of the result of the voting
- **Simplicity**: A good system should be understandable for as many people as possible, not only for experts. A simpler system probably also has less errors.

The aim of this thesis is, using techniques of cryptocurrencies, for example mixing services[1], to develop a system that allows the public to verify the correctness of the voting while preserving the voter's anonymity. Such mixing services can be used with Bitcoin[2] and are already a part of cryptocurrencies like Zcoin[1]. So we focus on the requirements of voter privacy and public verifiability.

1.1 Related Work

There are currently many projects in the field of e-voting. In particular, I looked at probably the most popular e-voting system of Switzerland, namely the one developed by the Swiss post[3]. This e-voting system roughly works as follows: Voters authenticate themselves before the voting starts and get a public/private key pair and a verification card containing a set of voter return codes linked to voting options, a voter confirmation value, a voter finalization value and a validity proof for such a finalization code. When the voter wants to submit its selected voting options, they get encrypted using its secret key and sent to a voting server, where they are validated. If the vote is accepted, it is added to a ballot box and a set of return codes is generated and returned to the voter. The voter then compares these return codes with the voter return codes received on the verification card. If the codes match, the voter confirms this by providing its confirmation value to the voting server. A finalization code is computed and sent back to the voting device. The voter checks whether the received finalization code matches the finalization value received during registration. If the verification is successful, the received finalization code serves as a confirmation of the correct submission and confirmation of the vote. To preserve voter privacy the system uses cryptographic mixnets that shuffle and re-encrypt/decrypt the votes multiple times. Only after that, the clear text votes are obtained. The system has auditors which verify the counting phase and is individually and globally verifiable.

The difference between the system developed in this thesis and the one above is firstly that the votes are not encrypted. This results in higher transparency and is less vulnerable. Another difference is that the mixing is not done by the voting authorities but by the voters themselves. This way, every voter is responsible for its own privacy. And another difference is that the mixing in the system described in this thesis is done before submitting the vote, while it is done after in the above system.

Chapter 2

Architecture

The system consists of a central server, the voters, and arbitrarily many monitors. While the server is run by an authority, which could be the government, in theory everyone can run its own monitor. In practice however a moderate amount of monitors is considered sufficient.
In a perfect world where everyone is honest, voters would just submit their selected voting options to the server and could be sure that they get counted as intended. While the designed system does assume the server to respond to requests, it does not assume it to always behave correctly. A malicious server could exclude some voters by just rejecting their votes for no reason. Or it could tell them the votes were accepted but actually not count them. If there is such bad behavior, it should at least be detectable. For this purpose there are the monitors. They reproduce the whole voting process and check whether the server acts correctly. If it does not, they can prove the server's misbehavior. Since everyone can run its own monitor, it is assumed that the voters trust a monitor. The fact that everybody can run its own version of the monitor and gather all data throughout the whole voting, makes the voting publicly verifiable.

To submit their vote, voters use a pair of a public and a secret key. Before the voting starts, the server publishes a list of public keys together with the identities of all eligible voters. A public key on that list is called *active* and every active public key is associated to exactly one eligible voter. The voters also get the corresponding secret key. Possession of a key pair where the public key is active allows a voter to successfully submit its vote.
Voters can change their keys by connecting to each other and create what is called a *transaction*. A transaction consists of two input public keys and two output public keys. Input and output public keys are also called *old* and *new* public keys, respectively. While the old public keys are the current active public keys of the clients, the new public keys are freshly generated. To ensure authenticity and integrity, a transaction is signed by both clients with the secret keys corresponding to the input public keys. A created transaction is sent to the server, where it is validated. If accepted, the transaction appends the new public keys to the server's list and removes the old ones from it. A public key

which gets removed from the list is called *inactive*[1] or *deactivated*. So after each transaction, the voters have a new public key on the list and also own the corresponding secret key. The goal of creating a transaction is to increase anonymity by not revealing which old public key is replaced by which new public key. The voters can create as many transactions as they want and the more they create the better their privacy is preserved.
After doing enough transactions, the latest key pair can be used to submit the vote to the server. The voter appends the currently active public key to the vote and signs it with the corresponding secret key. An accepted vote deletes the public key from the list and does not add a new one. The voter has finished its activity.

The server stores all the accepted transactions and votes in a hash chain. In a way this can be seen as the state of the system. As shown in Figure 2.1, the first hash in the chain goes over the initial list of public keys. Every following hash corresponds to either an accepted transaction or an accepted vote. It is computed over the previous hash together with the transaction or vote data. This hash chain has the benefit that transactions or votes which were added to the hash chain in the past cannot be modified anymore. By changing a transaction or a vote, the corresponding hash value changes too. And since every hash in the chain depends on the previous one, all further hashes in the chain change.

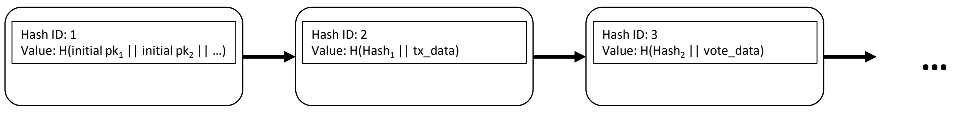

Hash ID: 1
Value: H(initial pk$_1$ || initial pk$_2$ || ...)
Hash ID: 2
Value: H(Hash$_1$ || tx_data)
Hash ID: 3
Value: H(Hash$_2$ || vote_data)
...

Figure 2.1: The hash chain maintained by the server. Every hash corresponds either to an accepted vote or an accepted transaction.

The monitor fetches from time to time the latest hashes in the hash chain as well as the latest transactions and votes from the server. It recomputes the hash chain and the public key list, which allows to reproduce the entire voting process. If the server acts maliciously or does not behave as it is supposed to, the monitor will notice and inform the public. The recomputed hash chain makes it easy to compare states between different monitors. The fact that the monitor rebuilds a copy of the voting state and gets all the data from the server, makes it possible for clients to let the monitor verify the server responses they get on transactions or votes. So there is always be a way for the clients to be certain about what happened to their submitted transactions or votes.
The monitors count all the votes received from the server and publish the state and the result of the voting.

[1] Note that "inactive" is not the same as "not active". An inactive key has been active at some point in the past. This does not necessarily hold for a not active key.

2.1 Voting Process

The voting process can be split up into four phases.

2.1.1 Setup Phase

Before the voting starts, the authority publishes the initial version of the public key list. The voters receive from the authority two files containing their public and secret keys. It would also be possible that the voters generate the initial keys themselves and register the keys at the authority. But we decided its easier to not have a registration and just let the authority generate and announce the keys. Additionally the authority distributes a file for the server's public key and a file with the voting questions and answer options. The server signs all its responses sent to the clients. This way the voters can be sure the message really comes from the server. It is important that the distribution of these files is secure. A compromised voter private key allows an adversary to vote on behalf of the real owner.

2.1.2 Transaction Phase

In this phase, voters can connect to each other and create a transaction. As mentioned earlier, on the initial version of the servers public key list, every eligible voter is listed with its ID and corresponding public key. So everyone can see which voter owns which public key. This allows to check for missing people and key duplicates. If voters just used their initial public key for voting, everyone could find out who voted for which options. Changing that is the goal of this phase and the idea behind transactions in general.

As mentioned in the first section of this chapter, a transaction replaces two old public keys by two new ones. The key idea of this replacement is that a transaction does not leak any information about which old public key is replaced by which new public key. Nobody but the participating voters knows the internal mapping. By looking at Figure 2.2, one can see that there are two possible ways to do the mapping, indicated by the solid and dotted arrows inside the box. Hence there are also two possible public keys that can be associated with each of the two voters after the transaction.

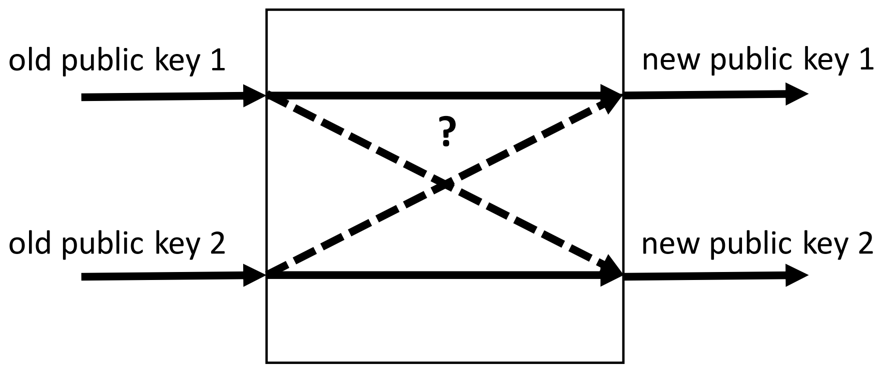


Figure 2.2: A transaction maps two old public keys to two new ones. Nobody except the voters who created the transaction knows which old one maps to which new one.

Looking at Figure 2.3, the reader can see that the uncertainty about which public key is owned by which voter spreads out exponentially by doing many more transactions. Voter A has two possible public keys after the first transaction. After the second layer of transactions, there are even four possible public keys, which can be traced back to voter A. The more transactions, the more possible public keys a voter has and the harder it gets to trace back a public key to a certain voter.
Every transaction the server accepts adds a new hash to its hash chain.

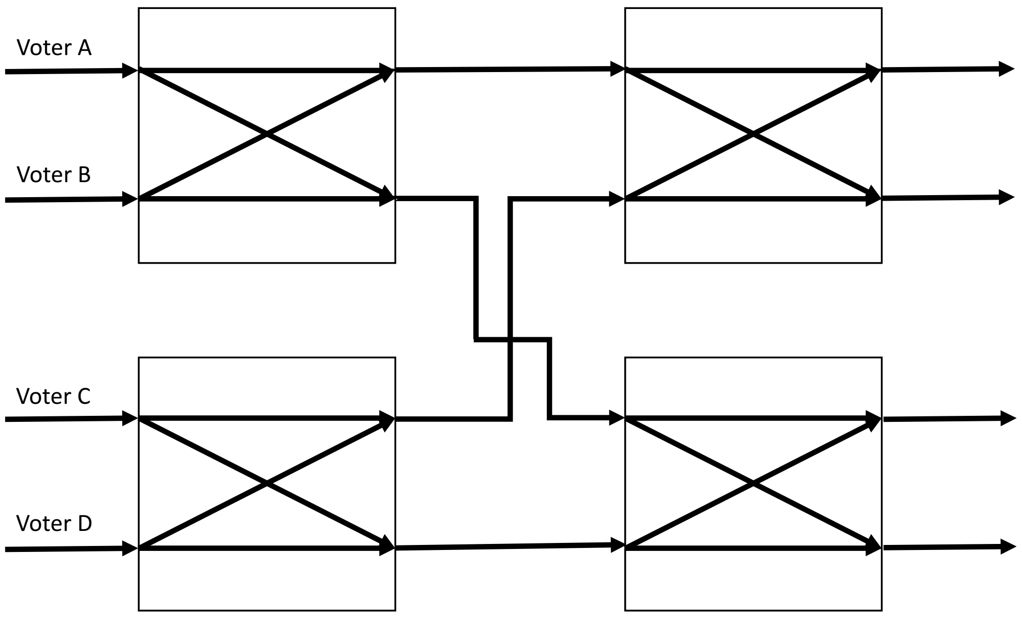


Figure 2.3: The uncertainty about which voter owns which public key spreads out by doing many transactions.

2.1.3 Voting Phase

When the voters decide they have done enough transactions and want to submit the vote, they sign it with the current secret key and send it together with the currently active public key to the server, where all votes are collected. The server validates the vote and decides whether to accept or reject it. Every voter can vote exactly once, and once the vote is submitted to the server, it cannot be changed anymore. After an accepted vote, the used public key is deleted from the list of active public keys.

2.1.4 Counting/Verifying Phase

During the whole process, the monitors constantly request the latest hashes, transactions and votes from the server. They verify the received data and rebuild the hash chain and the public key list. If anyone notices that the server acts maliciously, they stop requesting further data from the server, and inform the public about the server's misbehavior.
In addition to verifying, the monitors count all the votes received from the server and publish the results and the state of the voting.

Chapter 3

Implementation

The server as well as the monitor are both web applications and were implemented in Python using the microframework Flask[4]. Compared to other web frameworks in Python, for example Django, Flask has the advantage of being simpler and easier to learn, flexible and there are a lot of extensions. The client desktop application was developed with Electron[5], which uses Chromium and Node.js[6] and allows to write desktop applications using JavaScript, HTML and CSS. For the peer to peer communication, WebRTC[7] was used. To enable the peers to find each other, a signaling server is used which forwards the peer messages during connection setup. It was implemented using Node.js[6]. The Graphical User Interface of the client application was designed using Bootstrap[8] and jQuery[9]. How the different actors communicate is shown in Figure 3.1. All the communication is over HTTP. Data is transmitted in JSON format.

An overview of the developed system[1] and its actors is shown in Figure 3.1. The transaction window is used to create transactions with other clients, and the voting window is used to submit the votes. The server has different HTTP interfaces for receiving votes and transactions from the clients, and the monitors have corresponding interfaces where the client can get the server's response verified.

The figure also involves the concept of a recovery. During the Transaction Phase, it is possible that at some point the voter finds itself in a state of uncertainty, where it does not know which is the currently active key pair. An example would be when a transaction was sent to the server but the client crashes before receiving the response. To get out of this uncertainty state, a recovery can be started. The client keeps a history of its past transactions and in a recovery goes over the past transactions, and asks the server which was the last accepted transaction. From the new public keys of that transaction the client can get its currently active key.

[1]https://github.com/lucienschaller/eVotingSystem

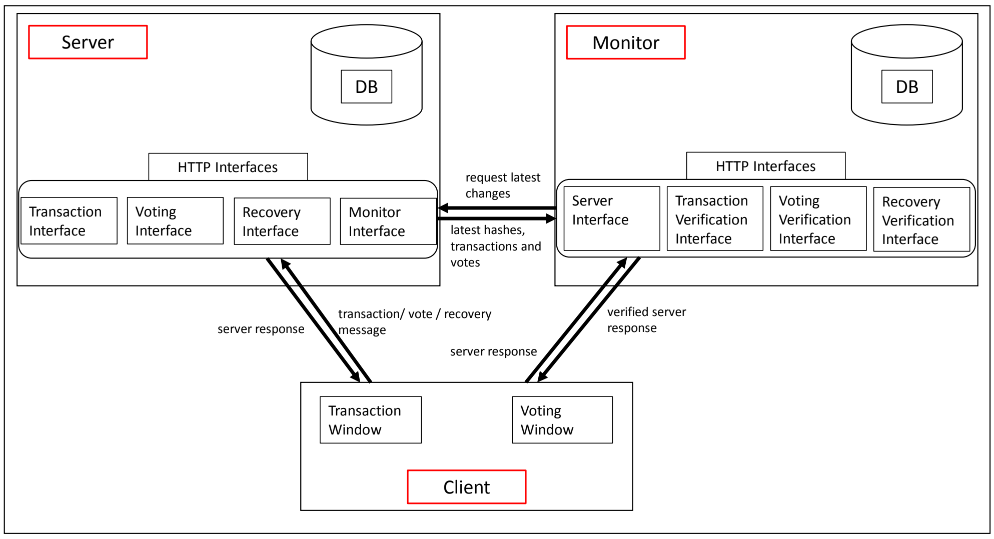


Figure 3.1: The different actors of the system and the interfaces over which they communicate.

3.1 Database

Both the server and the monitor use a MySQL[10] database to store accepted transactions, accepted votes and the hashes of the hash chain. The monitor additionally stores the errors appearing when the server behaves maliciously. Interaction with the database is done using Flask-SQLAlchemy's Object Relational Mapping. This allows to treat database records as Python objects.

3.2 Server

The server receives, validates and possibly executes transactions. It also collects the votes from the clients. Besides that, it helps voters to restore their currently active public key if they find themselves in a state of uncertainty. The server signs all its responses to the clients.

3.2.1 Receiving a Transaction

Upon receiving a transaction, the server first validates it. The procedure of verifying a transaction is depicted in Figure 3.2.

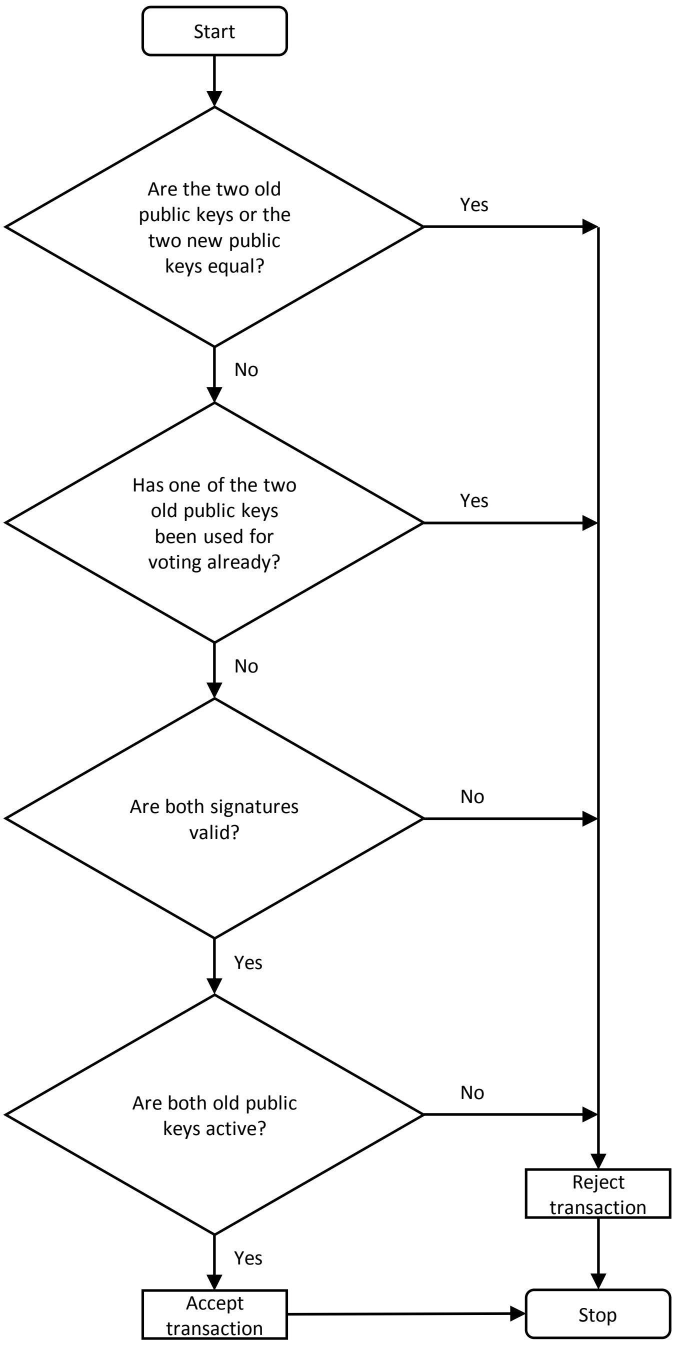

Start
Are the two old public keys or the two new public keys equal?
Yes
No
Has one of the two old public keys been used for voting already?
Yes
No
Are both signatures valid?
No
Yes
Are both old public keys active?
No
Yes
Reject transaction
Accept transaction
Stop

Figure 3.2: How the server validates a received transaction.

All the keys should be unique. If either the old or the new public keys are equal, there are two voters which own the same key pair. This can lead to multiple bad scenarios. For example, the first of the two owners which creates a valid

transaction will deactivate the public key. All following transactions from the other voter then are rejected because its public key is inactive. The voter is excluded from the voting process and there is no way to get back in. Even a recovery cannot help, because even if the active public key can be restored, the corresponding secret key is missing. No valid signature can be computed for future transactions without the secret key. Or if two voters have the same key and both want to submit their vote, the first one will get through, while the second one will get rejected because its public key has already voted and therefore also is not active anymore.
If the old public keys or the new ones are equal, the server rejects the transaction and sends back a response which indicates for each pair if the keys are equal or not.

If transactions were allowed to use as input a public key that has already voted, this would violate the essential voting condition that every voter should only be able to vote once. Normally, a voter sends the selected options for all questions together to the server. If the vote is accepted, the public key is deleted from the list. However if the voter does only submit a part of the selected options, the public key will not be deleted. A transaction could then replace the public key which partly voted with a new one, essentially giving a second vote to that voter. By rejecting transactions which include old public keys that have already voted, the above scenario can be prevented. If one of the input public keys already voted the server rejects the transaction and tells in the response which of the keys it was and gives a reference to the previous vote.

For a transaction to be valid, it is also necessary that both of its signatures are valid. This is to guarantee that the transaction really comes from the owners of the two old public keys. If at least one signature is not valid, the transaction is rejected. The server's response tells for each signature if it is valid or not.

Only active public keys are allowed as input public keys of a transaction. Without this condition it would be possible for an adversary which is not allowed to vote, to generate a key pair, which of course is not active, and turn it into an active key pair by creating a transaction. This would allow it to vote. So the condition ensures that only eligible voters can create transactions.
The condition also helps ensure that at any point in time (except after submitting the vote), a voter has only one active public key, and has therefore only one vote. By allowing public keys that are not active as transaction inputs, it is possible to turn any public key into an active one, which is equivalent to a vote. If at least one of the old public keys of the transaction is not active, the transaction is rejected. The server divides public keys that are not active into the ones which were deactivated by a previous transaction and those that were never active, called *invalid*. In the response, the server tells for each public key if it is active and if it is valid. For inactive public keys there is included a reference

to the deactivating transaction. The server filters out invalid keys because there could be adversaries, which are not eligible voters, but generate their own key and try to submit a transaction.

Finally, if all conditions are met, the transaction is considered valid and is executed. The list of public keys is updated, the old ones are replaced by the new ones. A new hash is computed and appended to the hash chain. The server sends back the newly generated hash and its ID.

3.2.2 Receiving a Vote

The server receives votes as messages that contain a list of triplets and the voter's active public key. Every triplet consists of two IDs and a signature. The IDs reference a question of the voting and the selected option for that particular question.

When receiving a vote message, the server first checks if the used public key is active. Firstly this ensures that only eligible voters can vote and secondly this prevents voters from voting multiple times because a public key gets deactivated after submitting a vote. If the public key is not active, the vote is rejected. The response depends on the reason for the public key not being active. A public key that is not active is either invalid, inactive or has already voted. If it was deactivated during a previous transaction, a reference to that transaction is included in the response. If it is invalid, there is no additional data included. And if the public key has already voted a reference to a previous vote is attached.
If the public key is active, the server then verifies every triplet. The vote is accepted only if all the triplets are valid, and rejected otherwise. The verifying procedure for a single triplet is shown in Figure 3.3.

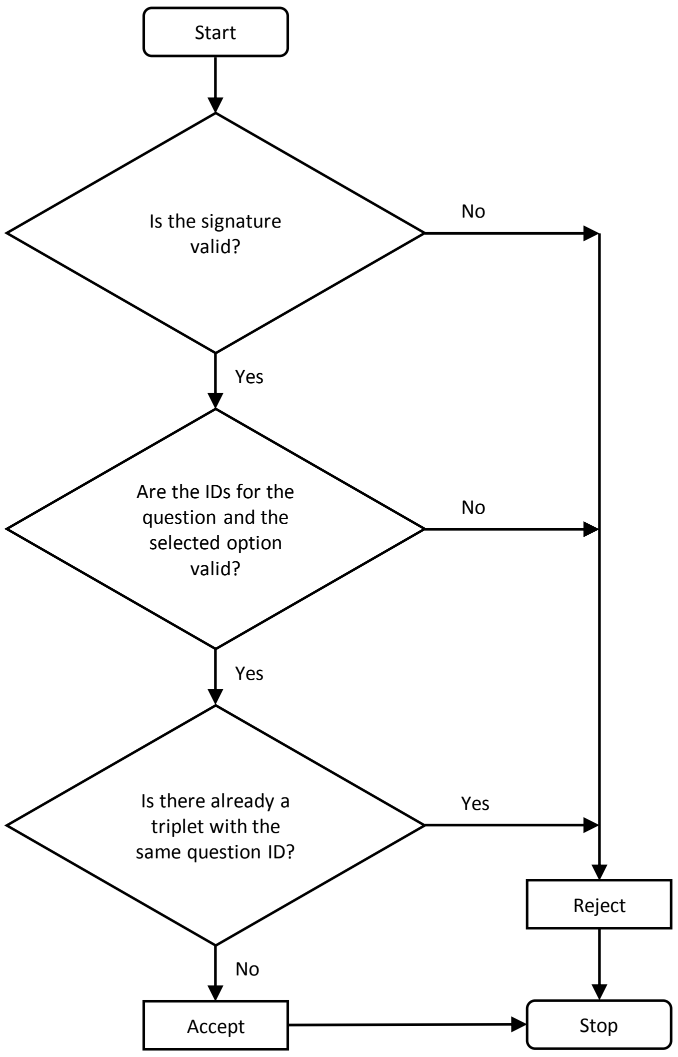

Start
Is the signature valid?
No
Yes
Are the IDs for the question and the selected option valid?
No
Yes
Is there already a triplet with the same question ID?
Yes
No
Reject
Accept
Stop

Figure 3.3: How the server validates a single triplet of a vote received from a voter.

A valid triplet has to contain a valid signature. This guarantees authenticity and integrity.
The IDs for the question and the chosen option have to correspond to real questions and options.
Every eligible voter can only choose one option for each question. This is why the server has to check that no two triplets have the same question ID.
Each triplet is stored as a separate record in the database and each triplet results in a new hash appended to the hash chain.

3.2.3 Receiving a Recovery Message

When the server receives a recovery message from a client, this message contains four public keys. The server looks for an accepted transaction which matches the four received keys. If it finds any, the corresponding hash and the hash's ID are returned to the client. In case no transaction is found, the response includes no additional data.

3.2.4 Monitor Interface

At a fixed time interval, the server receives requests from the monitor for the latest transactions, votes and the corresponding hashes. The requests contain the ID of the last hash which the monitor verified. The server responds with a fixed amount of consecutive hashes directly following the last verified one. Together with every hash the server sends the corresponding transaction or vote data.

3.3 Monitor

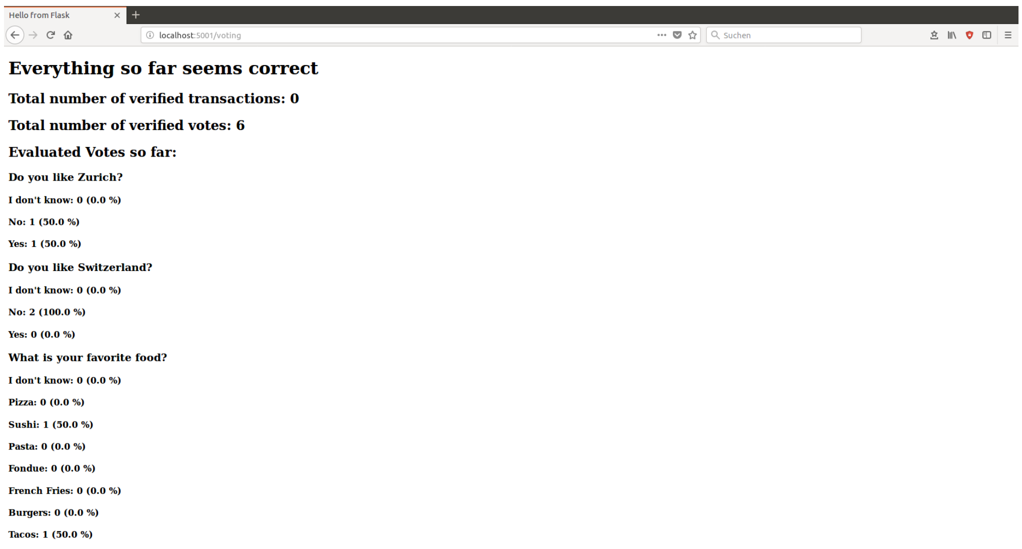
The monitor constantly fetches the latest hashes, transactions and votes from the server to reproduce the voting process and verify the server's behavior. In parallel, clients send the server's responses to their submitted transactions or votes to the monitor which verifies these responses for them.

3.3.1 Server Interface

When started, the monitor sends a single request to the server to get the initial list of public keys.
From then on, every two seconds the monitor sends a request to the server, asking for the latest hashes, transactions and votes. The requests contain the ID of the last hash verified by the monitor. For every received hash, it verifies the associated transaction or vote by running through the same verification procedure as the server, and updates its own hash chain. Because all the received transactions and votes were accepted by the server, the verification procedure should never end in a reject. If it does however, this is a proof that the server accepted a vote or a transaction which it should have rejected instead, hence a proof of misbehavior. Another proof of the server's misbehavior is a hash received from the server that does not match the hash computed by the monitor itself. Since the server signs the data it sends to the monitor, errors can without doubt be traced back to the server, and everyone can verify the signature.
Once the monitor detects bad behavior, it stops requesting new data from the server.

3.3.2 Counting

The submitted votes are evaluated by the monitor. It counts all the votes it receives from the server and serves an HTML page, where the current standings as well as the final results are published. Through that HTML page it also announces malicious server behavior. The HTML page is shown in Figure 3.4.



Everything so far seems correct

Total number of verified transactions: 0

Total number of verified votes: 6

Evaluated Votes so far:

Do you like Zurich?

I don't know: 0 (0.0 %)

No: 1 (50.0 %)

Yes: 1 (50.0 %)

Do you like Switzerland?

I don't know: 0 (0.0 %)

No: 2 (100.0 %)

Yes: 0 (0.0 %)

What is your favorite food?

I don't know: 0 (0.0 %)

Pizza: 0 (0.0 %)

Sushi: 1 (50.0 %)

Pasta: 0 (0.0 %)

Fondue: 0 (0.0 %)

French Fries: 0 (0.0 %)

Burgers: 0 (0.0 %)

Tacos: 1 (50.0 %)

Figure 3.4: The HTML page served by the monitor which shows the state of the voting.

3.3.3 Verifying Server Answers

When receiving a forwarded server response, the monitor first verifies the server signature. As mentioned earlier, the server signs all his responses to the clients. This signature makes sure that the server response which the monitor gets really originates from the server.
If the signature is invalid, the response cannot be guaranteed to come from the server so the monitor does not verify it. If the signature is valid, it tries to verify the response.
There are two types of server responses to transactions or votes. The first type are responses that reference other transactions or votes. An example is the response to a transaction which got rejected because one of the input public keys was previously deactivated. The second type are responses that do not reference other transactions or votes. A response to a transaction which was rejected because of an invalid signature would be such an example.

For server response of the first type the monitor checks that the referenced trans-

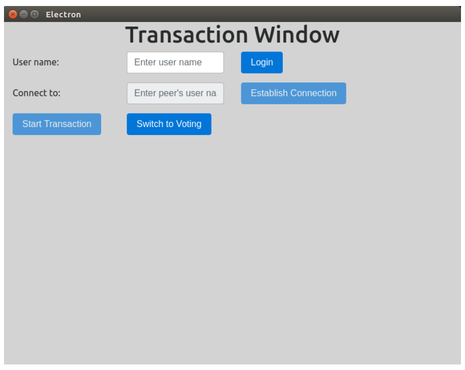
action or vote justifies the servers action. As an example, for a key that is claimed inactive due to a previous transaction it checks if the referenced transaction really deactivated that public key. There is one exception, namely for an accepted transaction. In this case, the monitor checks if its own hash chain contains a hash with the same ID and hash value as in the server's response, and if that hash really corresponds to a transaction which involves the four keys the client attached.
For server responses of the second type, the monitor just checks that there is no accepted transaction which matches the four received keys, or that there is no accepted vote from that voter.

For server responses of the first type, it can happen that the monitor has not yet received enough data from the server to be able to do the verification. For example if it has not yet received a transaction which is claimed to have deactivated a public key. In such cases the monitor does no verification and just waits for another request from the client.
In the answer to the client, the monitor tells whether the server signature was valid, whether it had enough data for the verification, and whether it was successful or not.

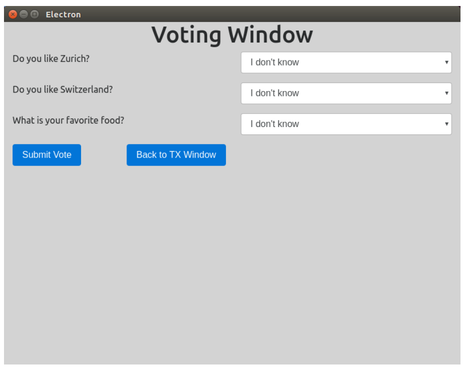
If the monitor can verify the server's response and detects bad behavior of the server, it stops requesting new data from the server and informs the public in its HTML page.

3.4 Client Desktop Application

The client application has two windows, between which the user can switch. One window is used to create transactions and one is used to submit the vote. Both of them are shown in Figure 3.5.



(a) The transaction window of the client application.



(b) The voting window of the client application.

Figure 3.5: Graphical User Interface of the Client Desktop Application

3.4.1 Transaction Window

In this window, the user can login and connect to other users to create transactions.
From the client's perspective the process of creating a transaction can be split up into three steps. In the first step the clients both login to the application and connect to each other. In the next step, the clients exchange the necessary public keys and signatures and send the transaction to the server. After they received the server's response, the connection between the clients is released and in a third step each of them individually contacts a monitor to verify the received response.

When the application is started, the transaction window is loaded and tries to connect to a signaling server. This signaling server is needed for the clients to find each other over the network. In the respective text field shown in Figure 3.5a, the user types in its desired user name and sends a login request to the signaling server by clicking the login button. If the connection attempt to the signaling server failed at the startup, now a notification shows up and the login request is dropped. The signaling server's answer to a login request is positive if the user name is not used already and negative otherwise. A notification tells the user whether the login was successful or not. Once logged in, the user can then type in the user name of the client to connect with. By clicking on the "establish connection" button, an invitation is sent to the signaling server which forwards it to the other client. If a client receives an invitation while it is already in a transaction, it is rejected silently. Otherwise a dialog shows up, asking the user whether to accept or reject the invitation. The answer is transmitted back to the first client via the signaling server. The invitation and response messages already contain the respective public keys for future use while creating the transaction. Once two clients are connected, only the one who sent the invitation can start the transaction by clicking on a button. That client is called the *initiator*, while the other client is called the *receiver*.

Once the initiator starts the transaction, the protocol shown in Figure 3.6 is executed to exchange the necessary data and submit the transaction to the server. All the messages exchanged between the clients are symmetrically encrypted. The initiator generates a new pair of public and secret key ($pk_{new,I}$, $sk_{new,I}$). Then it decides randomly for a key order by generating a random number and comparing it to a threshold. The server always replaces the first old public key of a transaction with the first new public key. So if for example the initiator's old and new public keys always were listed first, the mapping between old and new public keys would be leaked. It would be clear that the initiator owns the first one of the to new public keys after the transaction is executed. By choosing the order randomly, leaking the mapping can be avoided. Since the signatures for the transaction depend on the key order, the initiator then sends it together with the old public key $pk_{old,I}$ and the newly generated public key $pk_{new,I}$ to the receiver. The receiver itself also generates a new key pair ($pk_{new,R}$, $sk_{new,R}$). It computes its signature for the transaction and sends it together with its old and new public key back to the initiator which in turn also computes its signature now that it has all four keys. The initiator then sends a request containing the four keys in the determined order and the signatures to the server. As soon as it has the server's response, the response message is forwarded to the receiver. The server's signature ensures that the initiator cannot fool the receiver about what the server responded. As soon as both clients have the server's response, the connection between them is released, as no further interaction is needed.

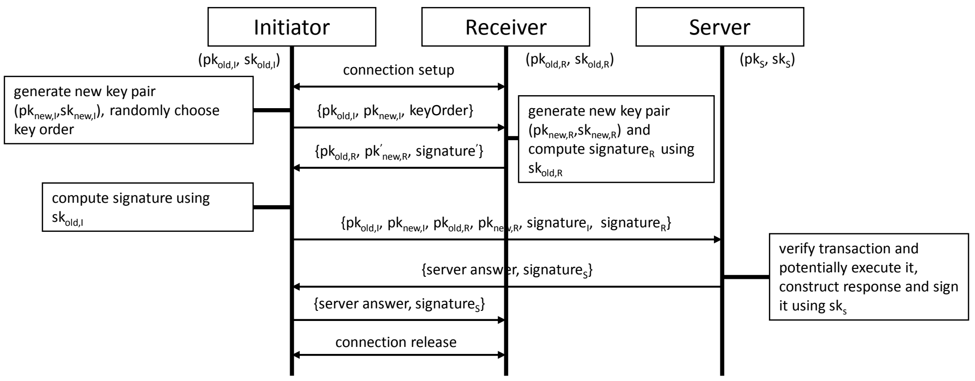

Initiator
Receiver
Server
($pk_{old,I}$, $sk_{old,I}$)
($pk_{old,R}$, $sk_{old,R}$)
(pk_S, sk_S)
connection setup
generate new key pair ($pk_{new,I}$,$sk_{new,I}$), randomly choose key order
{$pk_{old,I}$, $pk_{new,I}$, keyOrder}
generate new key pair ($pk_{new,R}$,$sk_{new,R}$) and compute $signature_R$ using $sk_{old,R}$
{$pk_{old,R}$, $pk'_{new,R}$, signature'}
compute signature using $sk_{old,I}$
{$pk_{old,I}$, $pk_{new,I}$, $pk_{old,R}$, $pk_{new,R}$, $signature_I$, $signature_R$}
verify transaction and potentially execute it, construct response and sign it using sk_s
{server answer, $signature_S$}
{server answer, $signature_S$}
connection release

Figure 3.6: The peer-to-peer protocol for creating a transaction.

In the following third step, each client first verifies the server signature. If the signature is invalid, the transaction is aborted and a recovery is started. If the signature is valid, the clients send the server response together with the four

keys used in the transaction to the monitor to let it verify the execution. If the monitor answers that the server's signature is not valid, the application is closed and a key recovery will start at the next startup. If the signature is valid but the monitor has not enough data to verify the server answer, the client waits a few seconds and asks the monitor again, up to five times. If after all five requests the monitor still could not verify the answer, the application closes. Again a recovery will start at the next startup. If the monitor could verify the server's response and no misbehavior of the server was detected, a notification tells the user if the transaction was accepted or why it was rejected. In case of an accepted transaction, the key files are updated with the new keys and the history of transactions is cleared. In case the transaction was rejected because the user's own old public key is inactive, a recovery is started to get back the currently active public key. In all other cases the notification gives the user advice about what to do.
If the monitor detects malicious server behavior during the verification, the client application notifies the user about it and closes afterwards.

3.4.2 Voting Window

When the voting window is loaded, the text file containing the questions and options is read from disk. The user then sees a grid of questions, each with a select box from which it can choose the desired option. By clicking on the submit button, all the select boxes are read out and turned into a list of triplets. These triplets each contain a question ID, an option ID and a signature computed with the voters current secret key. The list of triplets together with the voters current public key is then sent to the server. The response is forwarded together with the list of triplets and the public key to the monitor for verification.
If the monitor receives an invalid server signature, it does no verification and tells the user to submit its vote again. If the vote was indeed accepted the first time, the server will tell the client.
If the monitor has not enough data to verify, the client sends up to five requests. If after the fifth request the monitor still could not verify the response, the user is requested to submit its vote again. The server will tell if the votes were accepted the first time.
If the verification was successful, the user is notified whether its vote was accepted, or why it was rejected and what to do in that case.
If the monitor found an error during verification, the application closes after informing the user about the servers malicious behavior.

3.4.3 Recovery

As mentioned earlier, the client has the possibility to do a recovery if it does not know which is its currently active public key.

To even be able to do a recovery, the client needs to keep track of past transactions. This is achieved by storing for each transaction the four public keys as soon as the client knows them, together with the secret key corresponding to the potentially new public key. This list is stored in a text file on disk and read in as soon as the user logs in. If at this point the list is non empty, a recovery is started before the user can create new transactions. The list of past transactions is cleared once the public and private key of the client are updated. This happens either after a successful transaction or after a successful recovery.
When the application closes, intended or in case of a crash, the current list is written back to disk.

A recovery goes over the list in reversed order, starting with the four keys of the most recently created transaction. The four public keys are sent to the server. The server looks for an accepted transaction matching the keys. If it finds one, it adds the hash corresponding to that transaction and the ID of that hash to the response message. The server response is then forwarded to the monitor to let it verify.
If the server found a matching transaction and the monitor successfully verified it, the new active public and secret key of that transaction are written to the key files and the client can continue creating new transactions. If the monitor verified that there is no matching transaction, the next four keys of the list are sent to the server. If the recovery does not find a matching transaction for any of the keys in the list, the old keys are considered still active.
If the monitor receives an invalid server signature, the recovery is aborted and the application closed.
If the monitor could not do a verification because of a lack of data, the client sends up to five requests. If the monitor still could not do a verification after five requests, the recovery is aborted and the application closed.
If the monitor notices malicious behavior of the server during verification, it notifies the client. The user is shown a notification and the application closes.

3.4.4 Error Handling During a Transaction

During a transaction, several things can go wrong and potentially lead to uncertainty about what happened to a transaction. Most dominant are crashes of connections or actors and timeouts. For example if the server crashes after receiving a transaction, clients do not know whether the transaction was accepted or not. Therefore they are uncertain about which keys to use in the future. The same holds for a server timeout. Or if the transaction initiator crashes after receiving the signature and the public keys from the receiver. The receiver has no information about whether the initiator submitted the transaction or not. Such issues should be addressed.

Generally speaking, the process of creating a transaction can be split into three periods.
In the first period, the client knows for sure that the transaction has not been submitted to the server. This period starts for both clients as soon as they are connected and does not involve the server or the monitor.
During the second period the client is uncertain about whether the transaction was submitted or not. It starts for the receiver as soon as it sends its signature to the initiator, and for the initiator it starts as soon as the transaction message is sent out to the server.
In the third and last period the client knows for sure the transaction was submitted. It starts for both clients when they get the server answer. It is important to notice that these periods are partially different for the transaction initiator and the receiver.
Handling client crashes and timeouts depends on the period in which the error occurs. If there is a client crash or timeout during the first period, the other client simply aborts the transaction and goes back to the state before the connection. The old key is still active since no transaction could have been sent to the server.
If a client crashes or times out in the second period, the other client starts a recovery because it does not know if the transaction reached the server and if it was accepted.
In the third period, the connection between clients is released, so a crash or timeout of one client does not affect the other one.

In case of an unreachable server, the initiator shuts down. In case of a server timeout or other network error while contacting the server, the initiator aborts the transaction and starts a recovery. The receiver then will notice that the initiator closed the connection between them and also start a recovery.

In case of any error while contacting the monitor, the application closes since there is no way to verify a submitted transaction.

For unexpected errors like encrypting or decrypting errors or message parsing errors, the application generally is closed after showing a notification to the user.

3.5 Cryptography

For the cryptographic parts, PyCryptodome[11] was used on the server and monitor side, while Forge[12] was used on the client side. All keys are 2048 bit RSA keys. The signature scheme used is RSASSA-PKCS1-v1_5, together with a SHA256 hash function. The messages between the clients are encrypted using AES in GCM mode. The AES key is transmitted encrypted with RSA-OAEP using the public keys of the clients.

Chapter 4

Testing

The implementation is evaluated based on testing.

4.1 Server and Monitor

To test the server and the monitor, Python's `unittest`[13] module was used. A test case can be created by subclassing `unittest.Testcase`. To the test case, one can then add tests as functions. At the end of each such function, assertions can be used to check for the correct result. In addition to the test methods, one defines a `setUp()` and a `tearDown()` method, which are executed before and after each test function respectively.
To test the network interfaces of the server and the monitor, one can import their `create_app()` function into the test case. By calling that function a server or a monitor instance is created. By using the `test_client`[14] provided by Flask, HTTP requests can be sent to this instance and responses can be handled in any desired way.
The main difficulty was testing the exchange of data between the server and the monitor. To test this, one can use the `unittest.mock` library. This library makes it possible to replace certain parts of the system with mock objects and then make assertions about their usage. One can replace the request of the monitor, asking the server for the latest changes, with such a mock object. On this mock object one can then set the return value expected from the request. The system does not really send out the request but instead just takes the return value set on the mock object as if it was responded by the server.
To automate test execution, pytest[15] was used.

4.2 Desktop Application

The desktop application was tested without a library or an automated tool. One has thought of certain scenarios and reproduced those by hand. So these test cases tested not only the client application but rather the whole system. Crashes of the client can be simulated by either manually closing the application or calling close on the window via JavaScript. Timeouts on the server and on the monitor side can be achieved by using the `sleep()` function of the `time` module.

Chapter 5

Discussion

There are a few parts of the system that are rather inconvenient, and that are worth mentioning:

- As voters need to exchange some information to find together before every transaction, they both somehow know each other's real identity. This is a problem because the last voter with whom a transaction is created knows the identity and the public key used for voting afterwards. So it can easily find out what its peer voted for. Even worse, it can share that knowledge with the public and therefore completely expose the peer. With this in mind, a voter does best by doing the last transaction with someone trustworthy.
- Internal errors at the server side when submitting a transaction cannot be verified by the monitor, because the client and the monitor do not know where the server crashed and hence do not know if the transaction was executed or not.
- A transaction which has an invalid signature cannot really be verified by the monitor, since it does not know what message the server received and there is no old data that can be referenced for verification. In such a case, the client just sends multiple requests to the monitor which checks that no received transaction matches the four keys. But this is not waterproof since the transaction could arrive at the monitor just after the last request from the client. However this uncertainty would be cleared out on the next transaction, when the client's key then would be claimed inactive. A recovery would be started.

5.1 Future Work

There are several possible improvements one can think of:

- **Group Transactions**: One could modify the transaction part to allow not only transactions involving two voters, but also groups of more than two.

The uncertainty about which old public key maps to which new public key then gets even bigger. However this would require a new peer to peer protocol and would slow down the system because more coordination between the clients is needed.

- **Transaction Pool**: As mentioned above, voters have to somehow get in contact with each other outside of the voting system before they can create a transaction. They need to exchange user names. So it is a reasonable assumption that voters know the real identity of the voter with whom they create a transaction. This is a serious threat for anonymity. To circumvent it, one could design a *pool* where clients can connect to if they want to create a transaction. This pool then matches clients randomly. This way voters do not need to know anything from each other except the public key.
  One could still go a step further and use multiple pools controlled by different people, which would even increase the voter's anonymity more.

- **Voting Status**: One could improve the HTML page served by the monitor which shows the status of the voting. One could add some diagrams for better visualization and maybe show even more information.

5.2 Conclusion

In this thesis, an electronic voting system was developed which focuses on anonymity and transparency. For the most part, these goals were achieved, and the system can still be improved. The system's strengths are that it is easy to understand and uses no complicated nontransparent cryptographic tools. The voters are responsible for their own anonymity and can reproduce the whole voting process themselves by running their own monitor. The distributed nature of the system makes it difficult to control or influence large parts of the voting.

Bibliography

[1] Wikipedia entry: Cryptocurrency tumbler
https://en.wikipedia.org/wiki/Cryptocurrency_tumbler
Accessed: 05.03.2018.

[2] Website: CoinMixer, a Bitcoin mixing service
https://coinmixer.se/de/
Accessed: 05.03.2018.

[3] Website: E-Voting System of the Swiss post
https://www.post.ch/de/geschaeftlich/themen-a-z/branchenloesungen/e-voting-loesung-der-post
Accessed: 09.10.2017.

[4] Website: Documentation of the Flask microframework
http://flask.pocoo.org/
Accessed: 24.10.2017.

[5] Website: Electron
https://electronjs.org/
Accessed: 04.01.2018.

[6] Website: Node.js
https://nodejs.org/en/
Accessed: 04.01.2018.

[7] Website: WebRTC
https://webrtc.org/
Accessed: 28.12.2017.

[8] Website: Bootstrap's grid system
https://getbootstrap.com/docs/4.0/layout/grid/
Accessed: 09.02.2018.

[9] Website: jQuery
https://jquery.com/
Accessed: 09.02.2018.

[10] Website: MySQL
https://www.mysql.com/de/
Accessed: 24.10.2017.

[11] Website: PyCryptodome API
https://pycryptodome.readthedocs.io/en/latest/
Accessed: 30.12.2017.

[12] Website: JavaScript's Forge library
https://github.com/digitalbazaar/forge
Accessed: 31.12.2017.

[13] Website: Python's unittest module
https://docs.python.org/3/library/unittest.html
Accessed: 27.11.2017.

[14] Website: Flask's test client
http://flask.pocoo.org/docs/0.12/testing/
Accessed: 27.11.2017.

[15] Website: Pytest tool for automating test execution
https://docs.pytest.org/en/latest/
Accessed: 27.11.2017.