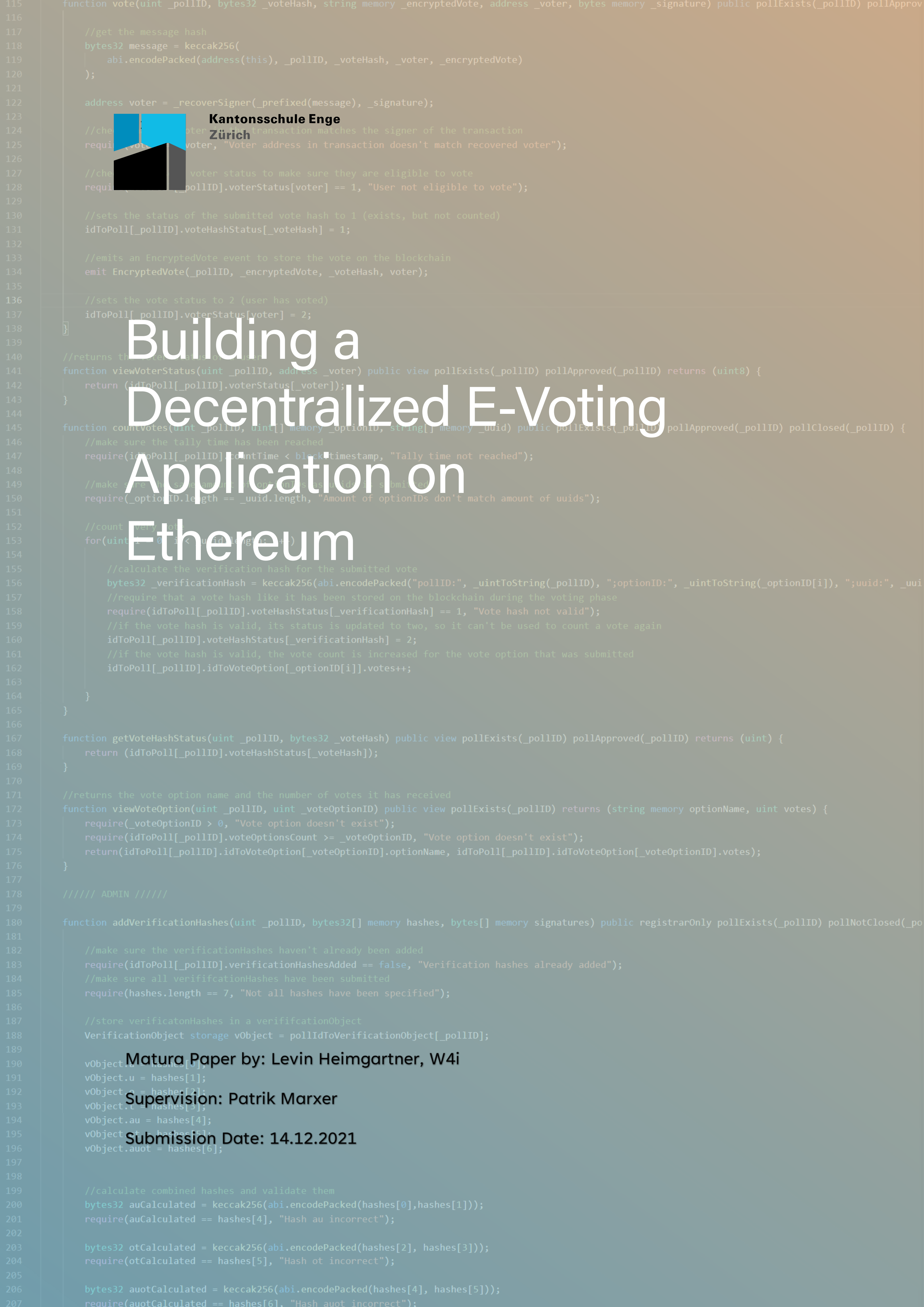
Kantonsschule Enge
Zürich

Building a Decentralized E-Voting Application on Ethereum



```
function vote(uint _pollID, bytes32 _voteHash, string memory _encryptedVote, address _voter, bytes memory _signature) public pollExists(_pollID) pollApprov

    //get the message hash
    bytes32 message = keccak256(
        abi.encodePacked(address(this), _pollID, _voteHash, _voter, _encryptedVote)
    );

    address voter = _recoverSigner(_prefixed(message), _signature);

    //check that the voter in the transaction matches the signer of the transaction
    require(_voter == voter, "Voter address in transaction doesn't match recovered voter");

    //check the voter status to make sure they are eligible to vote
    require(idToPoll[_pollID].voterStatus[voter] == 1, "User not eligible to vote");

    //sets the status of the submitted vote hash to 1 (exists, but not counted)
    idToPoll[_pollID].voteHashStatus[_voteHash] = 1;

    //emits an EncryptedVote event to store the vote on the blockchain
    emit EncryptedVote(_pollID, _encryptedVote, _voteHash, voter);

    //sets the vote status to 2 (user has voted)
    idToPoll[_pollID].voterStatus[voter] = 2;
}

//returns the voter status of a user
function viewVoterStatus(uint _pollID, address _voter) public view pollExists(_pollID) pollApproved(_pollID) returns (uint8) {
    return (idToPoll[_pollID].voterStatus[_voter]);
}

function countVotes(uint _pollID, uint[] memory _optionID, string[] memory _uuid) public pollExists(_pollID) pollApproved(_pollID) pollClosed(_pollID) {
    //make sure the tally time has been reached
    require(idToPoll[_pollID].countTime < block.timestamp, "Tally time not reached");

    //make sure the same amount of optionIDs as uuids is submitted
    require(_optionID.length == _uuid.length, "Amount of optionIDs don't match amount of uuids");

    //count every vote
    for(uint i = 0; i < _uuid.length; i++){

        //calculate the verification hash for the submitted vote
        bytes32 _verificationHash = keccak256(abi.encodePacked("pollID:", _uintToString(_pollID), ";optionID:", _uintToString(_optionID[i]), ";uuid:", _uui
        //require that a vote hash like it has been stored on the blockchain during the voting phase
        require(idToPoll[_pollID].voteHashStatus[_verificationHash] == 1, "Vote hash not valid");
        //if the vote hash is valid, its status is updated to two, so it can't be used to count a vote again
        idToPoll[_pollID].voteHashStatus[_verificationHash] = 2;
        //if the vote hash is valid, the vote count is increased for the vote option that was submitted
        idToPoll[_pollID].idToVoteOption[_optionID[i]].votes++;

    }
}

function getVoteHashStatus(uint _pollID, bytes32 _voteHash) public view pollExists(_pollID) pollApproved(_pollID) returns (uint) {
    return (idToPoll[_pollID].voteHashStatus[_voteHash]);
}

//returns the vote option name and the number of votes it has received
function viewVoteOption(uint _pollID, uint _voteOptionID) public view pollExists(_pollID) returns (string memory optionName, uint votes) {
    require(_voteOptionID > 0, "Vote option doesn't exist");
    require(idToPoll[_pollID].voteOptionsCount >= _voteOptionID, "Vote option doesn't exist");
    return(idToPoll[_pollID].idToVoteOption[_voteOptionID].optionName, idToPoll[_pollID].idToVoteOption[_voteOptionID].votes);
}

////// ADMIN //////

function addVerificationHashes(uint _pollID, bytes32[] memory hashes, bytes[] memory signatures) public registrarOnly pollExists(_pollID) pollNotClosed(_po

    //make sure the verificationHashes haven't already been added
    require(idToPoll[_pollID].verificationHashesAdded == false, "Verification hashes already added");
    //make sure all verififcationHashes have been submitted
    require(hashes.length == 7, "Not all hashes have been specified");

    //store verificatonHashes in a verififcationObject
    VerificationObject storage vObject = pollIdToVerificationObject[_pollID];

    vObject.a = hashes[0];
    vObject.u = hashes[1];
    vObject.o = hashes[2];
    vObject.t = hashes[3];
    vObject.au = hashes[4];
    vObject.ot = hashes[5];
    vObject.auot = hashes[6];


    //calculate combined hashes and validate them
    bytes32 auCalculated = keccak256(abi.encodePacked(hashes[0],hashes[1]));
    require(auCalculated == hashes[4], "Hash au incorrect");

    bytes32 otCalculated = keccak256(abi.encodePacked(hashes[2], hashes[3]));
    require(otCalculated == hashes[5], "Hash ot incorrect");

    bytes32 auotCalculated = keccak256(abi.encodePacked(hashes[4], hashes[5]));
    require(auotCalculated == hashes[6], "Hash auot incorrect");
```

Matura Paper by: Levin Heimgartner, W4i

Supervision: Patrik Marxer

Submission Date: 14.12.2021

Contents

Preface

When I was 12 years old, I first got interested in programming and was fascinated by the idea that I could just sit down on a computer and write an application or a website that could then be accessed by anyone in the world. I wrote my first application in swift for my iPhone and it didn’t do anything useful, it just allowed navigating between some pages on which I placed some text and buttons. I got the necessary knowledge to build the app from googling and watching YouTube videos. The next step major step in my coding journey was learning JavaScript, for which I bought a Udemy course to get started. Knowing JavaScript, I was then also able to program servers and complex websites, meaning I could build virtually any project I ever wanted to. In the years that followed I wrote websites, servers and bots for the numerous “business ideas” I had all the time. The problem was that I never actually completely finished any of the projects due to a loss of motivation or not having enough time to work on them. This was one of the reasons I decided to write my matura paper in computer science: I wanted to do a project with multiple software components that would be finished and useable at the end.

After having decided to write the paper in computer science I started going through a list of software-related business ideas for inspiration that I had in the last couple of years. I wasn’t convinced of most of them because they were too simple and, in most cases, would have just required a relatively simple website and server. At the same time, I by chance stumbled on the official Ethereum website, where I read the term “decentralized application” for the first time. Back then I had no idea what that was, so I started reading all the information about decentralized applications they had on the Ethereum website. After reading everything on their website and starting a coding tutorial for decentralized applications, I really liked the idea of writing my paper about that subject. It was also something new and challenging which I liked. The reason why I decided to write an e-voting system was that I was aware that there used to be e-voting trials in progress in Switzerland, but that they have temporarily been stopped because of security reasons. Additionally, a lot of websites and people said that decentralized applications could be used for e-voting, but there aren’t a lot of decentralized e-voting solutions actively used in the world.

At this point, I would like to thank my supervisor, Mr Marxer, for putting up with my project and enduring our numerous meetings that often lasted more than an hour and for providing valuable information and feedback. Then I would also like to thank my uncle, Mr Wihler, for taking the time to review the paper and giving me feedback on it.

1 Introduction

Since 2004, 15 cantons in Switzerland conducted over 300 trials with electronic voting systems. The trials were conducted within the project "Vote électronique", which is a project of the Swiss federal government and the cantons that aims to gradually introduce electronic voting in Switzerland. Up until 2019, the cantons had the option to choose between two e-voting systems: The system of the canton of Geneva and the system of the Swiss post. Things however changed in the summer of 2019, when first the canton of Geneva made their system permanently unavailable and shortly after the Swiss post announced that their system also won't be available anymore and that they are instead working on a new improved system. (Federal Chancellery, 2021) This was after multiple vulnerabilities in the post's system's source code were discovered after a public intrusion test conducted in 2019. (Wietlisbach, 2019) Since the summer of 2019, after the shutdown of the two systems, there are no e-voting systems available in Switzerland.

In 2008, when e-voting trials in Switzerland were already underway, the first blockchain was introduced. This blockchain powered the Bitcoin cryptocurrency and made Bitcoin the first cryptocurrency to solve the double-spending[2] problem without the need for a trusted authority or central server. (Wikipedia, 2021) With the launch of Ethereum in 2015, it became possible to also run applications on blockchains. These applications are called decentralized applications and are more transparent and, in most ways, also more secure than traditional centralized applications, which makes them interesting for e-voting.

With the rise of decentralized applications and the need for secure e-voting systems in Switzerland, the following question arises:

Is it possible to build a decentralized e-voting system that can be used for governmental elections in Switzerland on initiatives and referendums?

The goal of the paper is to write a decentralized e-voting system, whose core logic runs on an Ethereum based blockchain. The system should be able to conduct elections securely and transparently over initiatives and referendums like they are common in Switzerland. As far as the security measures in the areas of software and system design are concerned, the system should implement all the necessary security measures for carrying out a real election on a national level. The system however does not need to be production-ready, because the paper focuses on the design of the e-voting system with adequate security measures and the writing of the code. Not part of the paper is a production deployment that fulfils all the necessary security measures regarding the deployment and that has the capacity for carrying out a national or cantonal election. Also not part of the paper is the execution of steps that would be necessary for carrying out a real election on a national or a cantonal level, such as obtaining permits or finding a printing company that prints the access data for the voters so they can log in the e-voting system.

In the rest of the introduction, important concepts and definitions for the paper will be explained, including what decentralized applications are and some of the basics of cryptography. Building on the explanation of what decentralized applications are, the advantages of a decentralized voting system will be highlighted. At the end of the introduction, there will also be some

[2] https://en.wikipedia.org/wiki/Double-spending

additional requirements to the ones already mentioned before set out that the system should fulfil.

Subsequently, the system will be introduced in Chapter 2. An overview of different user types, the applications of the system and the steps of conducting an election with the e-voting system will be given. Additionally, it will be shown how the system is transparent and how this improves security.

In chapter 3, the different security measures that are implemented by the e-voting system will be shown and explained.

In chapter 4, the process of creating the paper and writing the code along with the problems I encountered during the process will be described.

In chapter 5, a technical overview of the system will be given where important steps of the election process will be explained in more detail. The chapter also gives a more technical overview of the admin levels and the software components that were already explained without technical details in the second chapter. Afterwards, the transactions fees (cost of using a blockchain) for carrying out an election will be shown and compared between two blockchains. If the requirements that were set out for the system, in the beginning, have been met will also be analysed and possible improvements for future versions of the e-voting system will also be shown.

Before starting with the paper some key concepts need to be explained first.

1.1 Encryption & Hashing

Encryption and hashing play a very important role on the internet we know today and is particularly important for electronic voting. In this chapter, some of the key principles of modern cryptography that are used in the voting system will be shortly introduced from a high-level perspective.

1.1.1 Public Key Cryptography

"Public key encryption, or public key cryptography, is a method of encrypting data with two different keys and making one of the keys, the public key, available for anyone to use. The other key is known as the private key." (Cloudflare, 2021) Anyone can encrypt a message with the publicly available public key, but the message can only be decrypted by the person that has access to the private key.

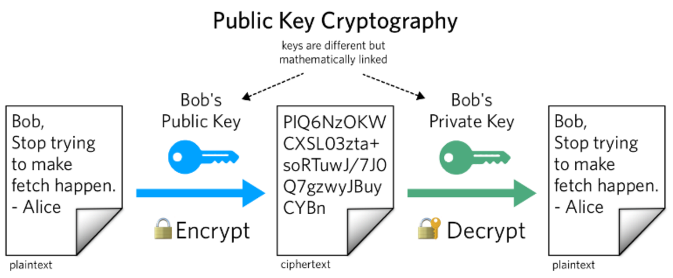

Public Key Cryptography
keys are different but mathematically linked
Bob's Public Key
Bob's Private Key
Bob, Stop trying to make fetch happen. - Alice
plaintext
Encrypt
PIQ6NzOKW CXSL03zta+ soRTuwJ/7J0 Q7gzwyJBuy CYBn
ciphertext
Decrypt
Bob, Stop trying to make fetch happen. - Alice
plaintext

Figure 1 Public Key Cryptography (Twilio, 2021)

1.1.2 Signatures

"A digital signature is a mathematical scheme for verifying the authenticity of digital messages or documents. A valid digital signature, where the prerequisites are satisfied, gives a recipient very strong reason to believe that the message was created by a known sender (authentication) and that the message was not altered in transit (integrity)." (Wikipedia, 2021) A signature can be made with a private key that only one entity has access to. The signature is mathematically linked to the publicly available public key, meaning that other entities can verify that the signature has been made by the entity having the private key, by checking if the signature belongs to the corresponding public key.

1.1.3 Hashing

Hashing is the process of converting a given key into another value. A hash function takes an input and outputs a newly generated value (hash) from the input according to a mathematical algorithm. (Edpresso Team, 2021) Modern hashing algorithms like Keccak256 only work in one direction, meaning that the input of the hash function can't be retrieved from the output of the hash function. The output of the hashing algorithm will almost all the time be different for different inputs. Even if for example only one letter of a message that gets put into a hash function is changed, the output will usually be completely different. However, it will always return the same output for the same input. The input of a hash function like Keccak256 can have any length, but the output of the hash function will always have the same fixed length, regardless of the input's length. Hashes are useful when you want to ensure that data hasn't been altered or that it is correct, without actually storing a copy of the data as a reference. Passwords are for example usually stored as hashes by websites, meaning that the website doesn't need to store the actual password to check that a user entered the correct password. The website can just compare the hash of the password entered by the user to the hash that it stored.

1.2 Decentralized Applications (Dapps)

To understand the paper, a fundamental understanding of how a decentralized application, short dapp, works and what it is are important. A dapp generally consists of two components. The first component is the client-side. The client-side usually is a website hosted on a conventional web server, but it can also be a mobile app on a phone or a desktop application. The client-side allows users of the dapp to interact with the other component of the dapp, the server-side. The server side of the dapp is a smart contract that runs on a peer-to-peer network of computers, the blockchain. The server side of the dapp is what differentiates a dapp from a conventional app.

1.2.1 Smart Contracts

Smart contracts are computer programs that are stored on a blockchain and can do essentially anything that most basic computer programs can do. (Ethereum Foundation, 2021) A smart contract is a collection of code, its functions, and the data it stores (the state of the application). A smart contract is a type of Ethereum account, but unlike normal Ethereum accounts, they are not controlled by a user but instead deployed to the blockchain. The developer of the smart contract can set rules inside the contract, which will then automatically be enforced by the code of the contract once it is deployed. Users can interact with the smart contract by submitting transactions that execute a function defined on the smart contract. (Ethereum Foundation, 2021) Users usually don't directly make a transaction to the smart contract, normally they interact with the smart contract over a client which makes the transaction.

1.2.2 Features of Dapps

Dapps have some unique features compared to conventional apps, because of the smart contract on their server-side.

Smart contracts protect the privacy of their users. This is because Ethereum is a pseudonymous network where transactions aren't publicly tied to the user's real identity, but instead to a unique cryptographic address. (Ethereum Foundation, 2021)

One of the biggest benefits of smart contracts is that the outcome of a smart contract is predictable because they always precisely execute based on the conditions written within their code. (Ethereum Foundation, 2021)

Most Ethereum smart contracts are written in solidity[3] or vyper[4]. But how does a smart contract look like? Below you can see a very basic smart contract written in solidity that has one function called *updateName()*, which updates the *name* string which is stored in the contract's state. Then there is also an address stored in the contract's state called *owner*. The first line of code inside the *updateName()* function requires that the user calling the function has the same address as the *owner* address. If the user calling the function has the same address as the *owner*, the *name* inside the contract's state will be updated to the *_newName* parameter of the function in the next line of code.

```solidity
// SPDX-License-Identifier: MIT
pragma solidity >=0.7.0 <0.9.0;

contract NameStorage {

    // The contract's state

    address public owner = 0xCa2215c9F029f2548C4964E1F84FD0131B6E3D83;
    string public name;

    // The contract's function

    function updateName(string memory _newName) public {

        // The next line of code checks if the sender of the transaction
        // has the same address as the owner. If this is not the case, the
        // transaction will be reverted, and the name won't be updated.
        require(msg.sender == owner);

        // Set the name in the contract's state equal
        // to the _newName parameter
        name = _newName;
    }
}
```
Listing 1 Example smart contract

When the contract is deployed, it is stored on every node of the blockchain. Afterwards, when someone makes a transaction to the contract and calls the *updateName()* function, every node on the network will execute the code inside the *updateName()* function with the same transaction data to ensure that the transaction is processed after the rules set out in the contract. The fact that hundreds of nodes owned by different entities verify transactions made to smart contracts, and the network only updates the state of the contract if the majority of the nodes deem the transaction to be correct, is what makes smart contracts so secure and remove the need for trust.

[3] https://soliditylang.org
[4] https://vyper.readthedocs.io/

Any interaction with a smart contract is irreversible and public. Anyone can view the state and the code of the contract and transactions made to it. Every transaction made to the contract or by the contract is stored in a public record. (Ethereum Foundation, 2021) For our simple contract, this means that everyone can view the current *name* and *owner* of the contract and a record of all transactions that have been made to the contract which includes all the changes that have been made to the *name* since the contract's existence.

Because the smart contract is publicly accessible, anyone can check the contents of the smart contract's code to view the logic of the contract. (Ethereum Foundation, 2021) This means that if someone for example orders the sample contract above from a developer and exclusively wants to have access to the *updateName()* function from their account, the buyer could easily verify that this is the case by looking at the contract's code.

The code of the smart contract written in solidity or vyper can't directly be deployed to the blockchain. For the Ethereum virtual machine to understand the code of the smart contract, it first needs to be compiled to bytecode by a compiler.

1.2.3 Gas and Transaction Fees

"Gas refers to the unit that measures the amount of computational effort required to execute specific operations on the Ethereum network. Since each Ethereum transaction requires computational resources to execute, each transaction requires a fee." (Ethereum Foundation, 2021) Gas is paid in the blockchain's native currency, which is Ether on Ethereum. The entity that makes the transaction bears these costs. In the case of the example smart contract in Listing 1, the owner would need to pay the transaction fee that occurs when changing the *name*.

Gas fees are necessary to help keep the Ethereum network secure. By requiring a fee for every computation executed on the network, bad actors are prevented from spamming the network. To avoid accidental or hostile infinite loops or other computational wastage in code, each transaction is required to set a limit to how many computational steps of code execution it can use. (Ethereum Foundation, 2021) The price of gas depends on the number of transactions sent to the network. Each block has a maximum gas limit and on average only every 15 seconds a new block is mined. This means that when a lot of transactions are sent to the network and want to be included inside a block, the gas price, i.e., the price to be included in the block, goes up because demand increases and supply stays the same. Since the London upgrade of the Ethereum main net in August 2021, blocks have variable gas limits up to a limit of 30 million units of gas and every block now has a base fee and a priority fee. The base gas fee needs to be paid for every transaction that wants to be included in the block, but the priority fee is optional. It is a tip that a miner receives additionally to the regular block reward, which means that miners are more likely to include transactions with high tips because they get more money and thus the transaction gets processed faster.

1.2.4 Mnemonics

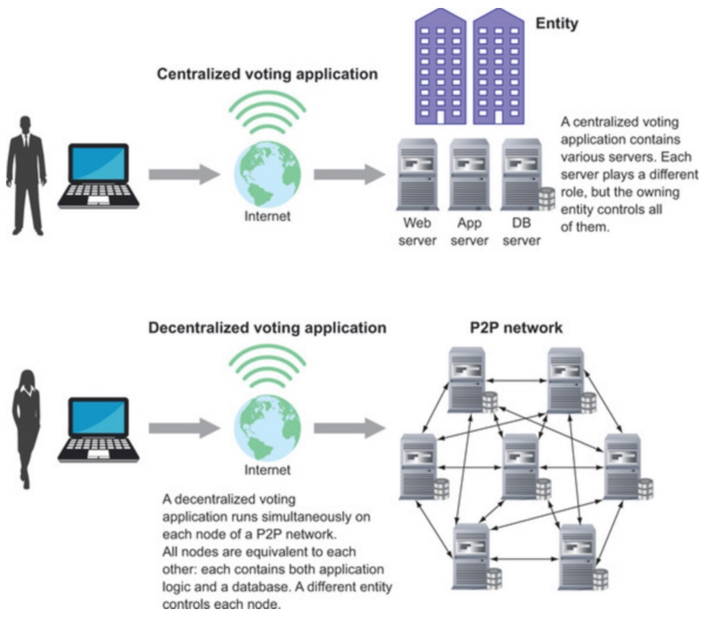
A mnemonic, also known as secret recovery phrase or seed phrase, is a set of typically either 12 or 24 words, which are used to derive the private key and thus also the address of a Bitcoin or Ethereum account. (My Crypto, 2021) Using a mnemonic makes storing the Ethereum account details easier because users don't need to write down the private key itself, where they might make spelling errors and subsequently lose access to their account. When users need to enter their account details in a cryptocurrency wallet or website to access their Bitcoin or Ethereum account this is much easier using a mnemonic where the users need to enter known words as if they needed to enter a private key.

1.3 Advantages of a Decentralized Voting System

A decentralized voting system has several advantages over a centralized voting system, which will be highlighted in this chapter.

First, we are going to look at a centralized voting system from a trust perspective. In a centralized voting application, the voters need to trust the organization that organizes the poll that they don't manipulate the outcome of the poll in some way. "It's easy to imagine, for example, that a malicious developer or administrator of the voting application, colluding with some party interested in a certain outcome of the voting, could access key parts of the system and tamper with the way votes are collected, processed, and stored at various levels of the application architecture. Depending on how the application has been designed, it could be possible for some malicious database administrators to even modify votes retroactively." (Infante, 2019)

From a security perspective, the voters also need to worry if the centralized voting system is adequately secured against external manipulation. "For example, external parties might be interested in having the voting go a certain way and might try to get their desired outcome by hacking into the system. [...] a centralized voting system includes only a certain number of servers located within the same network. Each server generally provides only one function, and it's, therefore, a single point of failure, not only from a processing point of view but also, and especially, from a security point of view. For example, if a hacker managed to alter code on the web server so that votes were intercepted and modified in that layer, the entire system would be compromised. The same outcome could be achieved by hacking only into the application server or, even better, into the database server. A breach of security in one part of the system is sufficient to compromise the security of the entire system." (Infante, 2019)


Centralized voting application
Entity
Internet
Web server
App server
DB server
A centralized voting application contains various servers. Each server plays a different role, but the owning entity controls all of them.
Decentralized voting application
P2P network
Internet
A decentralized voting application runs simultaneously on each node of a P2P network. All nodes are equivalent to each other: each contains both application logic and a database. A different entity controls each node.
Figure 2 Trust and security in Dapps thanks to P2P network replication (Infante, 2019)

As we can see there are some major security and trust concerns in centralized voting systems. Now we are going to look at a decentralized voting system which makes security and trust breaches mostly pointless because it runs on a blockchain. The blockchain removes the need for trust because if someone tried to maliciously alter a vote and propagate the modified vote to the rest of the network, the other nodes would detect the vote as modified during their validation and would reject it. They wouldn't store it in their local copy of the blockchain and wouldn't propagate the altered vote further throughout the network, so the malicious modification would become pointless. (Infante, 2019) The blockchain also makes the voting system more secure because even if a hacker managed to modify votes on one node of the blockchain network, the other nodes would spot and reject the alteration and wouldn't propagate it to the rest of the network. "Successful hacking would [...] require compromising not one server of the network but at least 51% of the nodes of the network simultaneously [...]." (Infante, 2019) This would be a very challenging and almost impossible task when the application runs on Ethereum because as of the 19th November 2021 the Ethereum blockchain has 3270 active nodes[5].

A decentralized voting system also has all the other benefits of a smart contract. It has no downtime, can be accessed anonymously, and all the transactions, its state and its code can be publicly accessed and verified. Even if a centralized voting system makes its source code public, the voters can't be sure that the code made available to the public will get deployed without any alterations. The code of the smart contract can also be looked at after it gets deployed since the code is propagated to every node on the network and can't be altered afterwards, so the voters can exactly see which code gets executed during the voting process. The counting of the votes will also be verified, and the result stored, on every node of the blockchain, meaning that every voter can verify that their vote has been cast and counted.

1.4 Requirements

Before starting to work on the project, I decided to set out some requirements for the voting system in addition to the requirements already mentioned in the Introduction. These requirements should also help me to know what the system should be able to do in the end.

To set out these requirements, I first analyzed a document called "Anforderungskatalog für eidgenössische Volksabstimmungen mit der elektronischen Stimmabgabe"[6]. The document was issued by the Swiss federal chancellery and contains information about getting a permit to carry out electronic elections in Switzerland including a catalogue with all the requirements that must be met to get a permit.

For this paper, I am only going to use section 6 of the catalogue called "Anforderungskatalog" because it contains all the requirements for an electronic voting system. The other sections that I am not going to look at mainly contain information about getting a permit for carrying out an election, which is not relevant for this paper because I am not going to carry out a real political election.

In section 6, I am going to look at sub section 6, which contains a table with all the requirements. Of all the requirements in the table, I am going to exclude the requirements about getting the authorization for carrying out an election and the evaluation of the election process. These are requirements 1 to 8 and 16 to 18. I am also going to exclude requirement 14, which determines how the results of an election need to be formatted. It is excluded because it also depends on the regular voting process by mail, which is also possible during public e-voting trial runs in Switzerland.

[5] https://etherscan.io/nodetracker
[6] (Federal Chancellery, 2014)

Catalogue Item	Requirement	Description
9	Individual verifiability	Voters must be able to detect whether their vote has been manipulated or intercepted on the user platform or during transmission.
10	Complete verifiability	It must be ensured that voters or the examiners have the opportunity to detect any manipulation that leads to a distortion of the result while preserving the secrecy of the vote.
11	User-friendliness	A system for electronic voting must be easy to use for voters and, in particular, should take into account the needs of as many voters as possible.
12	Voters with disabilities	The e-voting process shall be designed to accommodate the needs of eligible voters who are unable to cast their vote autonomously due to a disability.
13	Invalid ballots	The system should not provide for and accept invalid ballots.
15	Tallying of the votes	The results of an election need to be treated confidentially between the decryption of the votes and the publication of the results.

Table 1 Requirements that should be implemented in the voting system

Additionally, to the requirements set out in the catalogue, I also added some of my own requirements:

1. The system should also be able to be used in non-governmental elections, for example by a business for votes during shareholder meetings.
2. Important logic and data need to be processed and stored on the blockchain as far as possible, to make the election process transparent.

In the Requirement Analysis, it will be analysed if, and to what extend the requirements have been met.

2 User Perspective

This chapter gives a high-level overview of the voting system. First, the different user types are introduced and after that, an overview of the applications necessary for carrying out an election will be given. Then the major steps of the process of carrying out an election will be shown and explained. Finally, the transparency of the e-voting system from the perspective of users will be shown.

2.1 User Types

The voting system has multiple different user types, which will be shortly introduced in this chapter. First, there are the **voters** who can vote on polls. Then there are the **admins**, which are responsible for administrative tasks during the election process. There are 3 different types of admins, the **registrars**, the **chairpeople** and the **electoral board**, which all have different duties. The main duty of the electoral board is to oversee the election and to count the votes at the end of an election. The main duty of the registrars is to register voters and the main duty of the chairpeople is to create elections. Besides the voters and the admins, there is also **technical staff** necessary to deploy and maintain the code.

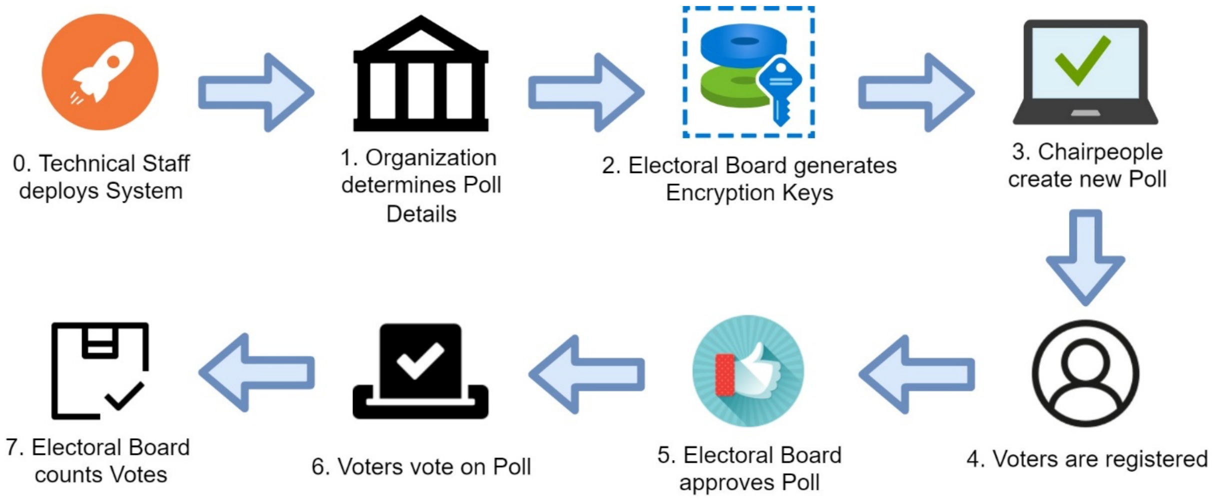
2.2 Application Overview

For the voting system to work, there are 6 different software components necessary, which will be briefly introduced in this chapter.

First, there is the **smart contract** which is responsible for enforcing important logic and storing important data like the polls and their details, votes or the voters. Then there is also a **CLI tool** that has multiple functions during the voting process, but its main function is to create an RSA key pair, whose public key is used to encrypt the votes, and the functionality to decrypt and count the votes at the end of an election. For the admins to interact with the smart contract, for example, to create a new poll, add voters or view the status of a poll, there is the **admin web app**. To cast their votes, voters use the **voting web app**. When logged in, the app displays the poll to vote on and the available options to vote for. After casting their vote, voters can also verify in the app that their vote has been cast successfully. To cast the voter's votes, there is a **relay** server necessary that forwards the votes to the smart contract and bears the transaction fee, so the voters don't have to. The **registration app** is a desktop application that allows the registrars to register voters, by creating access letters (also referred to as activation letters) for them, which can then be used to log in to the voting web app.

2.3 Election Process

There are multiple different steps when carrying out an election. A high-level view of these steps will be shown in this chapter, including the setup of the system and all preparations that go into conducting an election. These steps will later be explained again in more detail in the Product chapter.



0. Technical Staff deploys System
1. Organization determines Poll Details
2. Electoral Board generates Encryption Keys
3. Chairpeople create new Poll
4. Voters are registered
5. Electoral Board approves Poll
6. Voters vote on Poll
7. Electoral Board counts Votes

Figure 3 Election process

Step 0: The whole system gets deployed by the technical staff. This step is only necessary when the system is first used or changes are made to the code.

Step 1: The organization carrying out the election determines the details of the poll (title, opening & closing time etc.) and who should be eligible to vote on the poll.

Step 2: The encryption keys (public and private key) that will be used to encrypt the votes cast by voters are generated by the electoral board.

Step 3: The chairpeople create a new poll with the details that have been determined by the organization carrying out the election and the public key that has been generated in the previous step.

Step 4: The registrars create access letters with mnemonics for the people that are eligible to vote on the poll. The chairpeople then add the registered voters to the poll, verify them and confirm them.

Step 5: Before the voters can start to vote on the poll once the opening time is reached, the electoral board verifies that everything is correct with the poll. If this is the case, the electoral board approves the poll.

Step 6: Once the opening time is reached, voters can start voting on the poll.

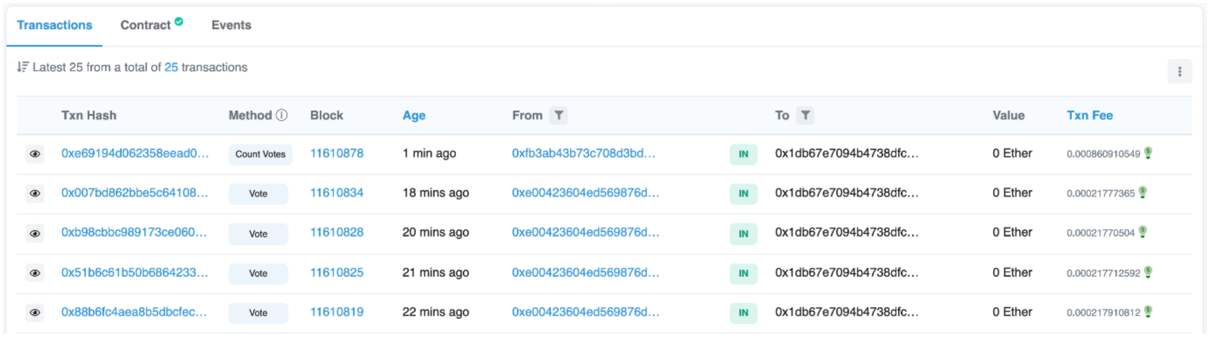
Step 7: When the tallying time has been reached, the electoral board uses the RSA private key that corresponds to the public key that was used to encrypt the votes, to decrypt and count the votes.

When another election needs to be made, the steps can be repeated in the same order, except that the system doesn't need to be redeployed every time. There can be multiple elections in progress at the same time.

2.4 Transparency

Being transparent and publicly accessible is a major benefit the decentralized e-voting system has over a centralized voting system because it runs on a blockchain. But what does transparency actually mean for the e-voting system? This question is answered in this chapter by showing the different ways transparency allows different entities to verify that the election isn't rigged and hasn't been compromised.

Everyone can look at the smart contract's code. The code of the smart contract is stored on the blockchain which is publicly accessible. To look at the smart contract's code entities can either run their own node, which will then receive a copy of the whole blockchain including the smart contract's code, the data it stores, event logs etc. or if they don't want to or don't know how to run their own node, they can visit a block explorer for the blockchain where the contract is stored. For the Ethereum main net, the explorer would be Etherscan[7], and for the Ropsten test network, the explorer would be Ropsten Etherscan[8]. On Etherscan the interested entities can then enter the address of the smart contract in the search bar, which will take them to a page with the contract's information. On this contract page on Etherscan, they can then view all the transactions that have been made to the contract under the "Transactions" tab.



Latest 25 from a total of 25 transactions

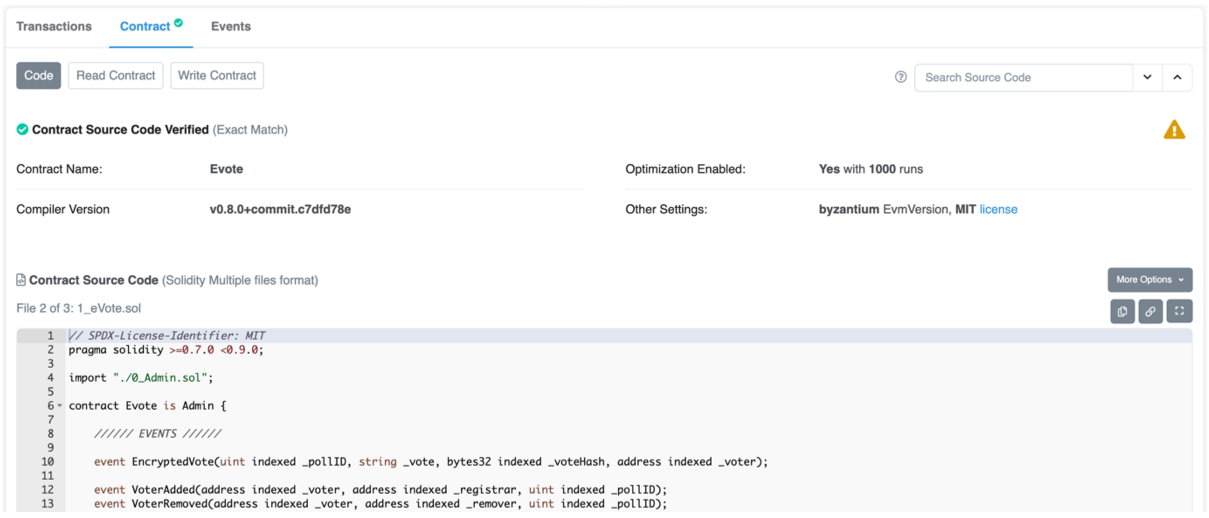
Txn Hash	Method	Block	Age	From		To	Value	Txn Fee
0xe69194d062358eead0...	Count Votes	11610878	1 min ago	0xfb3ab43b73c708d3bd...	IN	0x1db67e7094b4738dfc...	0 Ether	0.000860910549
0x007bd862bbe5c64108...	Vote	11610834	18 mins ago	0xe00423604ed569876d...	IN	0x1db67e7094b4738dfc...	0 Ether	0.00021777365
0xb98cbbc989173ce060...	Vote	11610828	20 mins ago	0xe00423604ed569876d...	IN	0x1db67e7094b4738dfc...	0 Ether	0.00021770504
0x51b6c61b50b6864233...	Vote	11610825	21 mins ago	0xe00423604ed569876d...	IN	0x1db67e7094b4738dfc...	0 Ether	0.000217712592
0x88b6fc4aea8b5dbcfec...	Vote	11610819	22 mins ago	0xe00423604ed569876d...	IN	0x1db67e7094b4738dfc...	0 Ether	0.000217910812

Figure 4 Ropsten Etherscan's Transaction tab of the Evote smart contract

As you can see in Figure 4, interested entities can view the transaction hash, the method that was called on the smart contract, the block in which the transaction was processed and when it

[7] https://etherscan.io/
[8] https://ropsten.etherscan.io/

was processed, who sent the transaction, the value of the transaction and the transaction fee in Ether. This means everyone can see when what action by whom is made on the contract. Everyone can see when votes are cast, a new poll is created, votes are counted, voters added etc.



Contract Source Code Verified (Exact Match)

Contract Name: Evote
Optimization Enabled: Yes with 1000 runs
Compiler Version v0.8.0+commit.c7dfd78e
Other Settings: byzantium EvmVersion, MIT license

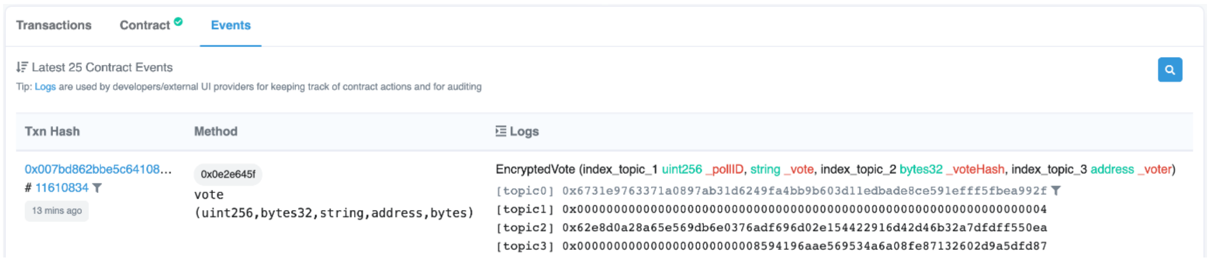
Contract Source Code (Solidity Multiple files format)

File 2 of 3: 1_eVote.sol

```
// SPDX-License-Identifier: MIT
pragma solidity >=0.7.0 <0.9.0;

import "./0_Admin.sol";

contract Evote is Admin {

    ////// EVENTS //////

    event EncryptedVote(uint indexed _pollID, string _vote, bytes32 indexed _voteHash, address indexed _voter);

    event VoterAdded(address indexed _voter, address indexed _registrar, uint indexed _pollID);
    event VoterRemoved(address indexed _voter, address indexed _remover, uint indexed _pollID);
```

Figure 5 Ropsten Etherscan's Contract tab of the Evote smart contract

In the “Contract” tab of Etherscan’s contract page, everyone can view the source code of the smart contract. It is important to note that this is the source code of the contract that is actually deployed to the blockchain and can’t be modified anymore, meaning that this is actually the code that handles transactions and stores data. People with a basic knowledge of smart contracts or IT will be able to verify that the source code is correct and has no security breaches and doesn’t have any backdoors for the government or other entities to manipulate elections. On this tab, all data can also be manually read from the contract, including data that by default is not displayed in the voting app, like the public key of a poll, the addresses of admins, if the vote of any voter has been received or counted and much more. Furthermore, voters can also look up the status of their vote to ensure that it really has been counted. They could even manually count their vote using Etherscan’s “Write Contract” function if they are worried that their vote hasn’t been counted.



Latest 25 Contract Events

Tip: Logs are used by developers/external UI providers for keeping track of contract actions and for auditing

Txn Hash	Method	Logs
0x007bd862bbe5c64108... # 11610834 13 mins ago	0x0e2e645f vote (uint256,bytes32,string,address,bytes)	EncryptedVote (index_topic_1 uint256 _pollID, string _vote, index_topic_2 bytes32 _voteHash, index_topic_3 address _voter) [topic0] 0x6731e9763371a0897ab31d6249fa4bb9b603d11edbade8ce591efff5fbea992f [topic1] 0x0000000000000000000000000000000000000000000000000000000000000004 [topic2] 0x62e8d0a28a65e569db6e0376adf696d02e154422916d42d46b32a7dfdff550ea [topic3] 0x0000000000000000000000008594196aae569534a6a08fe87132602d9a5dfd87

Figure 6 Ropsten Etherscan's Events tab of the Evote smart contract

Another important view of Etherscan is the Events tab. It shows the events that the smart contract emits. For the e-voting smart contract, the most important data that is shown here are the encrypted votes that the contract received and the voters that were added or removed from the contract. If a voter wants to verify if their vote really has been submitted additionally to the confirmation in the voting app, they can visit this page and search for the event that contains their vote and can then compare the data in the event with the vote confirmation they received after having voted in the voting app.

As you can see, every voter can verify if their vote has been cast and counted thanks to the transparency of the e-voting system. Furthermore, any entity can view the code of the contract, all transactions that are made to the contract and the data that is stored on the contract which includes the votes. Entities that run their own node can of course look at the same data as on Etherscan and can additionally even see how the individual transactions are processed, which of course requires more technical knowledge. The transparency of the system also means that any attempt at trying to manipulate the transaction would immediately be visible to all entities observing the election. If for example, someone unauthorized managed to remove some of the voters of a poll, which is highly unlikely because there are security measures in place to prevent this, this change would immediately be visible to anyone that is observing the poll because the contract's state will be changed in a transaction and events will be emitted when voters are removed, which will all be visible on Etherscan or a node. The poll organizer could then stop the election and set up a new one or deploy an updated smart contract where the security vulnerabilities have been fixed.

3 Security Measures

To ensure a secure election process, several security measures have been implemented into the voting system, besides the security features that are already given by the blockchain. These security measures will be shown in this chapter.

3.1 Encryption

This chapter covers encryption-related security measures in the e-voting system.

3.1.1 Vote Encryption

The voting system needs to keep the votes private until the election ends. To achieve this, the voting system uses an RSA key pair to keep votes private until the election ends. The public RSA key is stored on the smart contract and is used to encrypt the vote on the voter's devices before sending it to the blockchain where the vote is stored and publicly accessible. The RSA private key is itself encrypted using symmetric encryption during its generation by the electoral board and can only be decrypted with all the Ethereum private keys of all the electoral board members together. Requiring all the Ethereum private keys of the electoral board to decrypt the RSA private key ensures that not one member alone and no one that isn't authorized can decrypt the votes before the election ends to look at the results. In the case that one of the member's Ethereum private keys gets compromised this also ensures that no unauthorized person can decrypt the RSA key with only one of the electoral board members' Ethereum private keys.

The RSA keys are generated in the CLI tool using NodeJS' crypto module's *generateKeyPairSync()* function. In the code snippet below the options used to create the keys can be seen.

```
//create the keys with the NodeJS crypto module
const { publicKey, privateKey } = crypto.generateKeyPairSync("rsa", {
        modulusLength: 4096,
        publicKeyEncoding: {
            type: 'spki',
            format: 'pem'
        },
        privateKeyEncoding: {
            type: 'pkcs8',
            format: 'pem',
            cipher: 'aes-256-cbc',
            passphrase: passphrase
        }
    })
```
Listing 2 RSA key generation in the CLI tool

The private key is protected with a passphrase which is generated using the electoral board members' private keys. To generate the passphrase, the electoral board needs to enter their Ethereum private keys in the CLI tool. The first Ethereum private key entered is hashed using the Keccak512 hashing algorithm. This hash is then combined with the second Ethereum private key entered and hashed again. This process is repeated for every Ethereum private key. After the last Ethereum private key has been added to the hash of the previously hashed keys and hashed again, the hash that has then been generated will be used as the passphrase.

The CLI tool also returns a positions file along with the RSA keys as an output. The order in which the private keys are hashed to generate the passphrase is very important because a different order would result in a different hash as the final output. The positions file contains the addresses of the electoral board members in the order their Ethereum private keys have been hashed. This information then will be used by the CLI tool when the private key should be decrypted, and the passphrase needs to be generated again. The RSA private key along with the positions file is stored by the organization and the electoral board members until the election ends and the votes need to be counted. The RSA public key will be given to the chairpeople that then store it on the smart contract when they create the poll.

The order in which the keys are generated and where they are stored is summarized in Figure 7. The steps in Figure 7 take place in steps 2 and 3 of the Election Process.

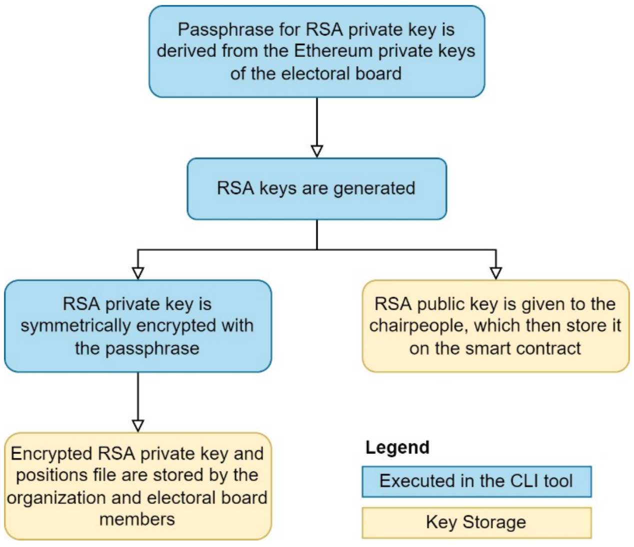

Passphrase for RSA private key is derived from the Ethereum private keys of the electoral board
RSA keys are generated
RSA private key is symmetrically encrypted with the passphrase
RSA public key is given to the chairpeople, which then store it on the smart contract
Encrypted RSA private key and positions file are stored by the organization and electoral board members
Legend
Executed in the CLI tool
Key Storage

Figure 7 RSA key generation and RSA key storage

Later in step 6 of the Election Process when a voter wants to submit their vote, the RSA public key that is stored on the smart contract is fetched and used to encrypt the vote. Additionally to encrypting the vote, the unencrypted vote is also hashed. All hashes and encrypted votes submitted by the voters will be unique because a salt is added to the vote before encryption and hashing. From the hashed vote and the encrypted vote along with some other information, a meta transaction[9] is created and signed with the voter's Ethereum private key. The vote is then cast with the meta transaction via the relay. The smart contract then verifies that the meta transaction is correct and sent by an eligible voter. The encrypted vote and the hash of the vote are then stored in the smart contract's event logs. The hash of the vote is additionally stored in the contract's state. The encrypted vote is stored only in the event log because the contract won't need access to it later and it is cheaper to store it this way. More information about storing the encrypted votes in the event log will be provided in the Storing the Votes chapter. The hash on the other hand needs to be accessed again later by the contract, which is why it is stored in the contract's state.

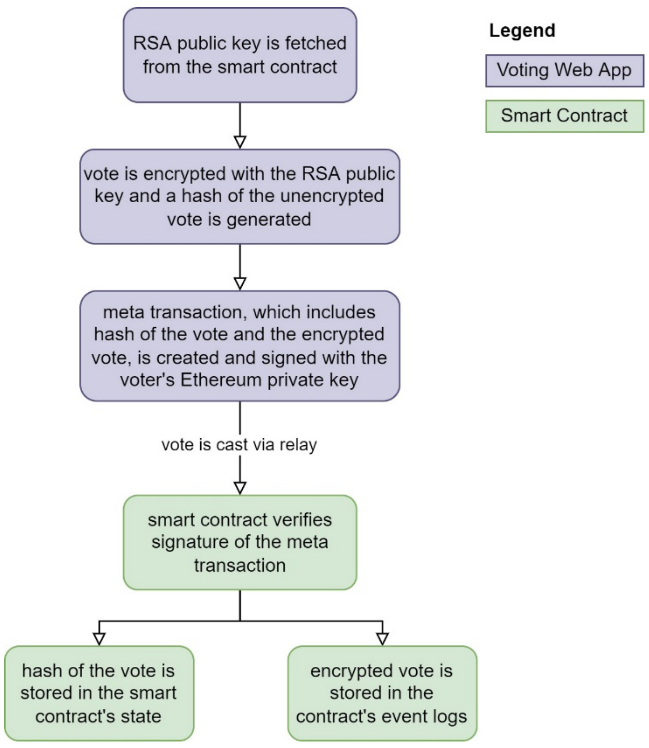
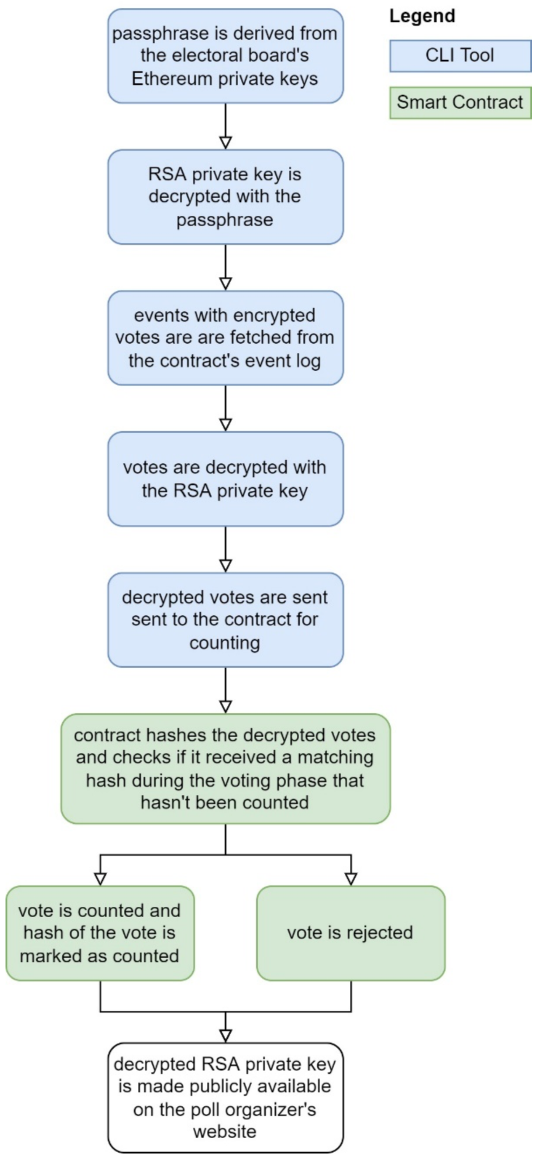

RSA public key is fetched from the smart contract
vote is encrypted with the RSA public key and a hash of the unencrypted vote is generated
meta transaction, which includes hash of the vote and the encrypted vote, is created and signed with the voter's Ethereum private key
vote is cast via relay
smart contract verifies signature of the meta transaction
hash of the vote is stored in the smart contract's state
encrypted vote is stored in the contract's event logs
Legend
Voting Web App
Smart Contract

Figure 8 Vote encryption and hashing in the voting app and storage on the smart contract

[9] Meta transactions are looked at in more detail in the Paying the Voters's Transaction Fees chapter

In step 7 of the Election Process, when the votes need to be decrypted and counted at the end of the election, the electoral board first uses the CLI tool to decrypt the RSA private key. The CLI tool does this by deriving the passphrase from the Ethereum private keys of the electoral board with the help of the information provided in the positions file. After decryption, the electoral board uses the CLI tool to count the votes. For this, the CLI tool gets all the events from the smart contract's event log that contain encrypted votes. The CLI tool then uses the decrypted RSA private key to decrypt the encrypted votes. The decrypted votes are then sent to the smart contract for counting. The smart contract regenerates the hashes of the votes that should be counted and checks if the same hash of a vote has been received during the user voting process. If this is the case, the smart contract counts the vote, and the hash of the vote is marked as counted in the contract's state to prevent double counting.

Comparing the hashes of the votes that are submitted for counting to the hashes that have been stored during the voting phase is a very important security measure. It ensures that votes that have been modified or weren't cast during the poll's voting phase aren't counted. If someone tries to submit a vote for counting that wasn't cast or has been modified, a hash for this vote will be calculated on the smart contract which won't match any of the vote hashes stored in the contract's state during the voting phase and the contract won't count the unrecognized vote.


Legend
CLI Tool
Smart Contract
passphrase is derived from the electoral board's Ethereum private keys
RSA private key is decrypted with the passphrase
events with encrypted votes are are fetched from the contract's event log
votes are decrypted with the RSA private key
decrypted votes are sent sent to the contract for counting
contract hashes the decrypted votes and checks if it received a matching hash during the voting phase that hasn't been counted
vote is counted and hash of the vote is marked as counted
vote is rejected
decrypted RSA private key is made publicly available on the poll organizer's website
Figure 9 Vote decryption and vote counting

After the votes have been counted by the admins, the decrypted RSA private key is made publicly available so the public can verify that the votes have been decrypted and counted correctly.

3.1.2 Encrypted Web Traffic

In a production deployment, the servers would use SSL and Transport Layer Security (TLS) 1.2 or 1.3 to encrypt web traffic between the different software components themselves and between the components and nodes, that can read data from the blockchain and make transactions. The table below shows which connections use encrypted web traffic and what data is exchanged in the connections.

Encrypted Connection	Exchanged Data
Frontend Server / Admin Web App	The admin web app is loaded from the server.
Frontend Server / Voting Web App	The voting web app is loaded from the server.
Admin Web App / Node	Data regarding the contract is read from the node and transactions are sent to the node.
Voting Web App / Node	Data regarding the contract, like the poll details, the vote options or the voter status, is requested from the node.
Voting Web App / Relay	The meta transaction is sent to the relay.
Relay / Node	Transactions that cast votes are sent to the node.
CLI Tool / Node	Registrar addresses, verification hashes and data necessary to count the votes are read from the node and transactions that count the votes are sent to the node.

Table 2 SSL/TLS encrypted connections and exchanged data

3.2 Verification Hashes

The verification hashes are generated by the registration app, after the voter accounts and access letters have been generated. The verification hashes are all the hashes of a hash tree that is generated from the output files of the registration app. This hash tree links the output together and provides security.

After the hash tree is generated, all the registrars are required to enter their Ethereum private keys in the registration app, which then uses the keys to sign the hashes of the hash tree to make sure they have been generated by someone authorized. If someone now modifies one of the files contained in the hash tree, the hash of the file won't be the same as before and the root of the hash tree won't be the same hash anymore, indicating that the files have been tampered with. Even if someone creates a new hash tree with the modified files, they wouldn't be able to sign the resulting output, because they don't have access to all the registrars' private keys.

The verification hashes allow the parties that receive one of the output files of the registration app to verify that the data they received hasn't been modified and is authentic. The verification hashes also link the different output files of the registration app together to provide additional security. How the verification hashes are generated and the importance of linking the files together will be shown in the next two sub chapters.

3.2.1 Verification Hash Generation

Before the verification hashes are generated, it is assumed that the contents of the output files of the registration app have already been generated.

Before the verification hashes are generated, some additional data is added to the output before hashing. This data includes the poll id, poll title and the address of the smart contract. The reason for this is to prevent replay attacks. If we assume that this data wouldn't be included in the output before hashing, the *users.json* and *addresses.json* files won't have anything inside them indicating for which poll they were generated, meaning that for example to a chairperson the users and addresses file and their hashes and signatures will mostly look fine, even though the data could be from an earlier poll, whose voter accounts could have been compromised.

The registration app has four different output files which will be part of the hash tree. The *addresses.json* file contains the Ethereum addresses of the voters that should be registered as voters while the *users.json* file contains the name and the physical address of the voters. The *one.zip* file contains all the first access letters of the voters and the *two.zip* file contains all the second access letters of the voters.

The users and addresses arrays that will be written to the *users.json* and *addresses.json* file, will receive a string as the first item of the arrays, which contains a nonce[10] in form of information about the poll and contract. After adding this information, the files are written to a temporary location.

```
"pollID=1;pollTitle=Sample Poll;contractAddress=0xA92B4d628Ebd96aEC48f163dfde8830Da390Da88"
```

Listing 3 Nonce string that is added to the registration app output

To each of the two folders containing the access letters, a file called *poll.txt* is added, which contains the same nonce string as in Listing 3, which is also included in the *addresses.json* and *users.json* file.

The *addresses.json* and *users.json* files directly get hashed, but the folders containing the first and second access letters and the *poll.txt* file first get compressed and the compressed versions then get hashed. The hashes of all files are generated using the Keccak256 hashing algorithm.

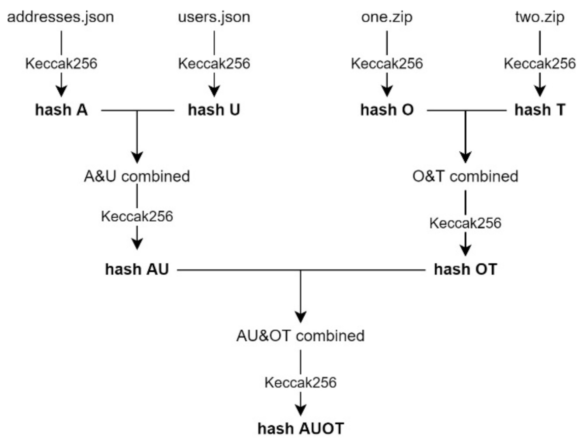


Figure 10 Hash tree of the registration app's outputs

After the verification hashes of the individual output files have been generated, two hashes always get combined and hashed together to create a hash tree as shown in Figure 10. Afterwards, the hashes get signed using the registrars' Ethereum private keys. The hash tree and the signatures are then placed in a file called *meta.json*, which is added to each of the output files.

3.2.2 Linking Voter Ethereum Addresses and Voter Identities

The Ethereum addresses and the real identity must be linked together in a way that preserves the privacy of the voters so that votes later can't be assigned to individual voters. The reason why the identities of the voters and Ethereum addresses must be linked together is that the admins need to be able to know that the Ethereum addresses they are going to add and approve as voters are the Ethereum addresses that belong to the voters in the *users.json* file. If the admins wouldn't have a way to know if the voters and Ethereum addresses correspond to each other and the output of the registration app wouldn't be hashed and signed, it would be possible for a rogue registrar to swap out the file containing the legitimate Ethereum addresses of the

[10] https://en.wikipedia.org/wiki/Cryptographic_nonce

voters, with a file only containing Ethereum addresses the rogue registrar has access to, before the file with the Ethereum addresses is sent to the chairpeople which would then add the compromised Ethereum addresses as voters.

The registration app prevents this exploit by first hashing the file containing the users and the file containing the addresses separately, and then hashing the hashes of those files together. This combined hash of the hashes of the two files links the two files together: When an admin receives a user and addresses file, they can calculate the hash of the two hashes of the files with the help of the CLI tool and compare it to the signed version of the combined hash in the meta file to check if the voter identities and the Ethereum addresses belong together. This also guarantees voter privacy, because a single Ethereum address cannot be assigned to a single voter, only all the voters can be assigned to all the Ethereum addresses. The verification hashes of course not only link the *users.json* and *addresses.json* files together but all the output files of the registration app.

3.3 Division of Power

Much like in a state of law, there is a division of powers in the smart contract among three admin levels. The goal of the division of power among the admin groups is to provide for checks between the groups and prevent the concentration of unchecked power, which could potentially be abused by some of the admins. In this chapter, the security measures that are part of or related to the division of power will be shown.

3.3.1 Two-Thirds Majority Logic

The contract requires a two-thirds majority of the electoral board, chairpeople or chairpeople and registrars combined to successfully execute some important actions, like approving polls or adding and removing admins. The two-thirds majority should prevent a concentration of power inside the admin group and the abuse of power by individual admins.

3.3.2 Two-Thirds Request Cooldown

After an admin creates a new two-thirds request, there is a 5-minute cooldown, during which the same admin can’t create a new two-thirds request. Because only one two-thirds request can be open at a time, an open two-thirds request will be overwritten when a new one is created. The cooldown is a security measure that should prevent a rogue admin or an unauthorized person that has gained access to an admin account, from blocking all the two-thirds requests that should remove their account from the admins or from blocking any other action that requires a two-thirds request, by constantly creating new two-thirds requests that overwrite the request that is currently open.

3.3.3 Complete Admins

For the admins to be able to execute functions only they have access to, their admin group needs to be complete. This forces the admins to always be complete to effectively use the contract and conduct elections. Only one admin can be missing of all the admins when the admins are being updated, meaning that at any time when an admin is missing, the admin group that needs to add a new admin to the incomplete admin group, is complete and can do so.

3.3.4 Checks and Approvals

Before a poll can start, first the voters and then the entire poll needs to be reviewed and approved. In these approvals, one admin type always reviews and approves the work of a different admin type. The poll is created by the chairpeople and the voter accounts then are generated by the registrars, which then are reviewed and approved by the chairpeople. After that, the whole poll is reviewed and approved by the electoral board.

3.4 Anonymity

The voting system provides voter anonymity by using Ethereum addresses that are not linked to the voters' personalities. To ensure that the Ethereum addresses aren't linked to the voters' real identities during account creation, the voters and their Ethereum addresses are split from one another, and the addresses are then randomly mixed before being saved. Additionally, the mnemonic that is necessary to login to the voting app and to derive the Ethereum address and the Ethereum private key is split, which also ensures that the votes can't be linked to the voters' real identity to ensure the anonymity of the voters and it also improves security.

The registration app splits the mnemonic into two parts. The full mnemonic consists of 12 words, which is split in the middle into two times 6 words. The first access letter receives the first 6 words and the second access letter the last 6 words. The access letters are then sent to two different facilities for printing. One facility prints and distributes the first access letters, and the other facility prints and distributes the second access letters.

The reason why the mnemonic gets split into two parts is that in the case someone unauthorized gets hold of one of the access letters, they won't be able to vote on the poll because to do so, they would need both parts of the mnemonic and thus both access letters. This is similar to how credit cards and their pins are sent in separate letters because it provides additional security.

3.5 Preventing Replay Attacks[11] in Meta Transactions

In the voting system, two points could theoretically be attacked by replay attacks. The first point that could theoretically be attacked is the meta transaction. If an Ethereum account would be reused as a voter account across multiple instances of the e-vote contract that could potentially be deployed on a blockchain, someone could get hold of the voter's vote and signature that they cast on a poll on contract A. This bad actor could then use the vote the voter cast on contract A and also cast it on contract B because the signature would still be accepted. To prevent this, the contract's address is included in the signed transaction that casts the vote, so the signature is only valid on the smart contract the vote was meant for. To prevent replay attacks of votes between polls in the same contract, the poll id is also included in the signed transaction, so that a vote is only valid on the poll it was originally meant for. Together the smart contract address and the poll id act as a nonce inside the signed transaction to prevent replay attacks. The second point that could be attacked is the verification hashes, which we already looked at in the Verification Hashes chapter.

4 Process

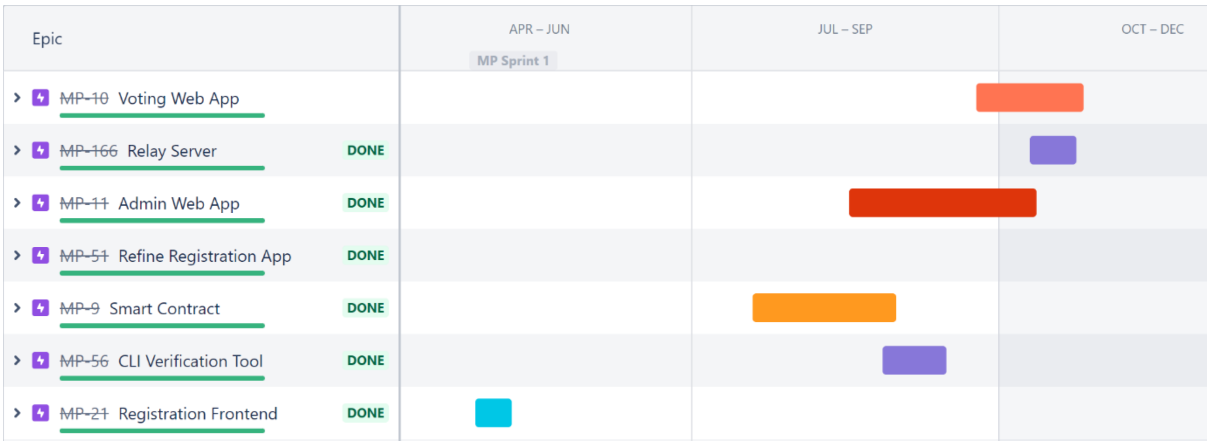
This chapter covers the process of planning the system's design and writing the code of the different components, the major problems I encountered during the process and how I fixed them. It also covers the research I did to learn about decentralized applications, solidity and the blockchain.

4.1 Time Planning

I made my original time plan in Jira Software, which is a tool for planning software projects and keeping track of tasks and bugs. Because I did not know in advance how much time I would have available every week, I decided to make a broad time plan that did not contain fixed planned tasks for every week, but instead, I planned periods for every software component, during which the first version of the entire software component should be written and finished at the end of it. This made me more flexible and didn't require me to always update my time

[11] https://en.wikipedia.org/wiki/Replay_attack

plan, which would have been the case if I had planned multiple smaller goals and steps that were supposed to be done at fixed dates.


Epic

APR – JUN

MP Sprint 1

JUL – SEP

OCT – DEC

Epic	
~~MP-10~~ Voting Web App	
~~MP-166~~ Relay Server	DONE
~~MP-11~~ Admin Web App	DONE
~~MP-51~~ Refine Registration App	DONE
~~MP-9~~ Smart Contract	DONE
~~MP-56~~ CLI Verification Tool	DONE
~~MP-21~~ Registration Frontend	DONE

Figure 11 Screenshot of Jira Software's roadmap view with some of the major tasks

I wrote the first version of the registration app during spring vacation 2021. I started with the registration app because it is mostly independent of the other software components and the rest of the system's design wasn't complete back then. Also during spring vacation, I finished most of the design the final system should have.

During summer vacation I first wrote the smart contract because it is the core component of the system and to start working on the other software components, I first needed the ABI[13] of the finished smart contract.

At the end of summer vacation, I needed to do a significant reschedule, where I swapped the order in which the admin app, voting app and the CLI tool should be finished. Originally, I planned to first write the voting app, then the admin app and finally the CLI tool, which turned out to be a mistake, because to be able to test the software, the components needed to be written in reversed order than originally planned, which is also the order in which they are necessary during the election process.

I wrote the CLI tool, admin app and voting app between summer vacation and autumn vacation and made some refinements to all three during autumn vacation. It was also during autumn vacation when I decided not to use the Gas Station Network[14] (GSN) for meta transactions but instead write my own relay. Writing the relay itself was not a big deal, however, the smart contract and the voting app also needed to be updated to support meta transactions.

After I finished most of the software and started the writing process, I used Notion[15] instead of Jira Software for time planning. This is because Jira Software is primarily for planning software projects and keeping track of software-related tasks and bugs that need to be addressed.

4.2 Research

When I started working on this paper, I didn't know much about decentralized applications, let alone the programming of smart contracts. To get started, I first read the definitions and explanations of smart contracts and dapps on the official Ethereum website[16] to learn more

[13] https://en.wikipedia.org/wiki/Application_binary_interface
[14] https://opengsn.org/
[15] https://notion.so
[16] https://ethereum.org

about them. On their website I also found a link to a "Learn by coding" tutorial called Crypto Zombies[17], where you learn to build a Zombie game that runs on a smart contract. The tutorial was a lot of fun and it taught me the basics of solidity, but I was still missing a lot of knowledge, especially regarding how to set up a development environment on my laptop.

I searched the web for books about smart contracts and dapp development and read some reviews and then decided to buy three books. The first book that I purchased was "Building Ethereum Dapps" by Roberto Infante. I liked the book because it not only covers how to write smart contracts and how to set up a development environment but also covers how the Ethereum blockchain works and what benefits decentralized applications have. However, since the book was written in 2019 there were a lot of changes made to the solidity programming language and some of the tools that were used in the book have since been deprecated. During the coding process of the smart contract, I mostly relied on this book and the official solidity documentation[18]. When I got stuck, I often used the Ethereum stack exchange[19] or Medium[20] articles to find a solution to my problem.

The other two books that I bought were "Ethereum for Web Developers" by Santiago Palladino and "Blockchain in Action" by Bina Ramamurthy. I primarily relied on Roberto Infante's book because the information it contains about writing smart contracts are the best in my opinion, which is why I didn't use the other two books as much. I primarily used them for advanced reading and learning more about topics like meta transactions or distributed storage solutions like IPFS[21].

4.3 System Design Problems

This chapter covers the two major problems I ran into while designing the system. The first problem was where the votes should be stored and the second problem was how the voters' transaction fees that occur when casting a vote should be paid.

4.3.1 Storing the Votes

The first major problem I ran into when planning the system was how the votes should be stored. The obvious first thought I had was to store the votes directly in the contract's state. But this approach had one big problem: Storing data on the blockchain is very expensive, especially when it's a lot of data. To solve this problem, I worked out three possible solutions. The first solution was to store the votes centrally on a server. This solution would have taken away some of the key features and benefits of a decentralized voting system, which is transparency and the security given through the blockchain and decentralisation and would have resulted in the system being quite similar to a traditional voting system. The second solution was not to use a blockchain to store the votes but instead distributed storage like IPFS. This solution was good because transparency and some of the benefits of decentralisation would still be given to a certain extent, while not costing as much as storing the data on the blockchain. The third solution however was even better. In this solution, the votes are stored in the smart contract's event logs, which is a lot cheaper than storing it directly in the contract's state and it doesn't decrease the security of the voting system. It also keeps the system transparent and maintains all the benefits of decentralisation and the blockchain. Storing data in events is cheaper because

[17] https://cryptozombies.io/
[18] https://docs.soliditylang.org/
[19] https://ethereum.stackexchange.com/
[20] https://medium.com/
[21] https://ipfs.io

the state of the smart contract isn't changed, also meaning that the data can't later be directly accessed by the smart contract.

4.3.2 Paying the Voters' Transaction Fees

On Ethereum every user needs to have sufficient funds (Ether) on their account if they want to execute state-changing functions on contracts, like the voting system's smart contract. This is necessary to cover the transaction fee.

This is a problem for the voting system because it means that voters would need to pay money to buy Ether to cast their vote. For one, you can't require people to pay with their own money to vote on an election and you also can't trust an average citizen to know how to buy Ether and transfer it to their e-voting account. One option to avoid this problem would be to send the Ether required to cast the vote to the accounts of the voters, so they don't need to use their own money to buy Ether, however, this option has some major flaws. Gas prices change from one second to another, which means that some voters might not be able to cast their vote, because of too high gas prices. Most voters will only need some of the Ether to cast their vote if gas prices are low when they cast their vote, which means that more money would be spent by the election organizer than necessary. Another problem is that some voters could exchange the Ether for real money instead of using it to vote and the Ether sent to voter accounts that won't be used will often be permanently lost.

For the reasons stated above, the option of sending the Ether to voter accounts, would be very expensive and inefficient and not feasible for the voting system.

After searching for some time and comparing different options, I found another way for users to interact with smart contracts, without having to pay for their own transactions and it is called "meta transactions". The way how a meta transaction works is, that the voter hashes the transaction and then uses their account's private key to sign the hash of the transaction, generating a signature. The signature and the transaction, which are now referred to as the meta transaction, are then sent to a relay, which uses an Ethereum account with enough funds on it to create and pay for a new transaction that contains the meta transaction. When a contract receives the transaction, the contract can verify that the meta transaction has been sent by the original user. To do this, the contract hashes the transaction and then checks if the signature corresponds to the original sender's Ethereum address. If this is the case, the contract executes the requested function, which may be a money transfer, or like in our case, the casting of a vote.

There are multiple ways to implement meta transactions in a smart contract. One project is the Gas Station Network (GSN), which is a global network of relays that allows any contract to implement meta transactions.

I chose not to use the GSN and instead wrote my own relay. This was for a couple of reasons. I only needed to use meta transactions for one function and implementing GSN support into the project would have been rather complicated and time-consuming. Also, the GSN isn't available on all Ethereum based blockchains, which would have made me less flexible in choosing a blockchain to deploy the smart contract to.

4.4 Coding

For all the components of the voting system, I used git as the versioning tool and Bitbucket[22] as a remote. To keep track of the issues I was currently working on and for time planning, I used Jira Software[23], which also nicely integrates with Bitbucket. I used visual studio code[24] as my primary code editor. This chapter shows the process within the different software components, the major problems I encountered and how I fixed them.

4.4.1 Registration App

When I started working on the registration app, I was still fine-tuning the final design of the voting system. I wasn't sure whether it should be a web app or a desktop application. I decided to start working on a web application that has a *react*[25] frontend and an *express*[26] server as a backend because this setup could easily be converted to a desktop app at a later point. To create the react app I used *Create-React-App*[27]. Another important package was *react router*[28], which I used for client-side routing, and *reactstrap*[29] to bootstrap the frontend's user interface.

After having finished the frontend and the server, I also finished designing the system and concluded that for security reasons it makes more sense for the registration app to be a desktop application. This is because it is more secure when the voter accounts are generated on an offline computer. I created the required files and installed the necessary packages to make the project work with *electron*[30], which allows building cross-platform desktop apps with JavaScript. At this point, the frontend of the electron app still communicated with the backend over HTTP, which is neither very effective nor very secure. Electron has a feature called inter-process communication, which allows the frontend to directly communicate with the backend, which I decided to implement. Besides inter-process communication, I also needed to make some other small changes to make the app work with electron, like using electron's API to store temporary data. After having finished this first version of the registration app, I made the code more readable and created a build of the registration app for macOS.

Later I came back to make some bug fixes and improvements to the registration app. This most notably included fixing a bug that used wrong data types to create a hash tree of the application's outputs and adding a nonce to the outputs. I also added the feature to specify a custom poll description and the option to add a link to the voting website that aren't hardcoded and will be printed on the access letters.

When I finally wanted to build the app for Linux using the *electron-forge*[31] build and packaging tool, the packager always failed with an error. I tried to fix the error returned by *electron-forge* but without success. I then tried *electron-packager*[32] as the build tool, along with *electron-installer-debian*[33] for packaging, which worked without any problems.

[22] https://bitbucket.org
[23] https://www.atlassian.com/software/jira
[24] https://code.visualstudio.com
[25] https://reactjs.org
[26] https://expressjs.com
[27] https://create-react-app.dev/
[28] https://reactrouter.com/
[29] https://reactstrap.github.io/
[30] https://www.electronjs.org
[31] https://www.electronforge.io/
[32] https://github.com/electron/electron-packager
[33] https://www.npmjs.com/package/electron-installer-debian

4.4.2 Smart Contract

I used the truffle framework to build and test the smart contract. For testing, I additionally used a mock blockchain called ganache[34], which mocks the behaviour of a real blockchain but runs locally, processes transactions almost instantly and allows the developer to easily customize most of the blockchain's settings.

Before starting to work on the smart contract, I created a checklist with multiple small steps that I wanted to follow during the coding process. Every step also included a series of automated tests which I was going to write in JavaScript using the mocha[35] test framework which is built into truffle.

For the first steps, I followed my checklist as planned and wrote a series of automated tests after every step to make sure my code worked. However, I quickly realized that my procedure wasn't very efficient. I usually spent about 3 times as much time writing the tests as actually writing the smart contract code, because the tests often needed to be adapted after every step, which took a lot of time. So I decided to stop updating the automated tests after every step and only write them when a feature of the contract is finished. After introducing this change, I progressed a lot faster than before.

I continued to follow my steps in order and manually tested the code after each step, but I decided to skip 2 of the steps and complete them later.

The first step that I skipped, was the implementation of the code inside the function that verifies that the hash tree and the signatures of the verification hashes are correct. I initially wrote the code, but I couldn't get it to work properly because the smart contract's function returned different results for the recovered signatures and the calculated hashes than the registration app, so I decided to leave it and come back to it later. When I later returned to the code and tinkered with it for a while, I realised that the code inside the smart contract worked correctly and the code in the registration app was incorrect. The registration app calculated the hash of two hashes using string as a data type, but the data type of the hashes inside the smart contract is bytes32. I updated the function that creates the hash of two hashes and the signatures inside the registration app, so it converts the input hashes to bytes32 before hashing and creating the signatures.

The second step that I skipped was the implementation of Gas Station Network support. The reason why I skipped it and how I fixed the issue that arose with it, was already shown in the chapter Paying the Voters's Transaction Fees.

During the coding process, I decided to divide the contract into 3 smaller contracts in separate files. The contracts are connected through an inheritance chain. The highest contract in the chain is the *TwoThirdMajority* contract that stores all the data and contains all the logic that is necessary for the two-thirds majority logic to work. It doesn't inherit from another contract. The second contract is the *Admin* contract which inherits from the *TwoThirdMajority* contract, whose code it requires to create admin add and removal requests. It stores the different admins, contains all the logic to add and remove admins and has modifiers that restrict access of some functions to certain admin levels. The last contract in the inheritance chain is the *Evote* contract and it inherits the *Admin* contract. The *Evote* contract stores all the data related to polls and

[34] https://trufflesuite.com/ganache/
[35] https://mochajs.org/

contains all the code that is related to polls, like creating a new poll, adding verification hashes, casting votes, or counting votes.

After having finished all the steps, excluding the ones that were skipped, I started working on the test cases. I wrote 3 different files containing tests. The first file tests all the modifiers inside the *Admin* and *Evote* contract. The second file tests all the code inside the *Admin* contract excluding the modifiers and the third one tests all the code inside the *Evote* contract. The code inside the *TwoThirdMajority* is tested inside the tests of the *Evote* and the *Admin* contract because its core functions are private and only internally accessed inside the code of the *Evote* and the *Admin* contract.

The tests inside the individual test files are grouped into bigger groups and then often inside those groups again into smaller groups. I tried to write at least one positive and one negative test for each function and modifier of the contract.

When I first started writing the tests, I didn't know a good way to test time-dependent logic. Because of this, I first didn't write any tests concerning the parts of the smart contract that are responsible for time and commented the corresponding lines in the contract out, so I could finish writing the other tests. After some googling, I found a very good medium article about time-dependent tests in truffle by Paul Razvan Berg[36], which helped me solve the issue by changing the block timestamps.

After running the automated tests, I deployed an instance of the contract to the Ropsten test blockchain, so I could manually test it along with the admin app, the voting app and the CLI tool. Later when I completed the smart contract development steps that I skipped in the beginning, I updated the tests accordingly to also test the new code. I also added CircleCI[37] support to automatically test the smart contract's code when a new version is committed to Bitbucket.

4.4.3 Admin App

I wrote the admin app with react and created the project using Create-React-App. The most important packages I used besides react, were *ethers*[47] for communication with the smart contract, react router for client-side routing and reactstrap for bootstrapping the user interface of the app.

To manage the Ethereum private keys of the admins that interact with the contract, I decided to use Metamask[48]. Metamask is a browser extension that makes Ethereum account management easier because the private key only needs to be entered once. Metamask injects a web3 provider in the browser window which then is used by the admin app to make transactions and read data from the blockchain. In development, Metamask is especially useful because it can store multiple Ethereum accounts and allows to switch between them.

I started with writing the login component and logic of the admin app and afterwards started working on the poll overview page, the create new poll page and the poll detail view. After having finished the components related to the polls, I moved on to the admin page. While writing the first version of the project, my goal was to make the app work at its core and leave out some less important features like automatic page reloads when data on the smart contract is updated.

[36] (Berg, 2019)
[37] https://circleci.com
[47] https://docs.ethers.io/
[48] https://metamask.io

Later when I returned to the admin app to make some improvements, there was one issue in particular, I wanted to fix. When I wrote the first version of the admin app, every major component had its own code that communicated directly with the smart contract to get the data it requires. This meant that the app got the same data multiple times because different components requested the same data. Also, the code that logs the user in was spread over two components. I wrote two custom hooks[49] to tackle these problems. The first hook is called *useUserManagement* and it contains all the code that is responsible for logging admins in and out and getting the admin's user level. In its state, it also stores the ethers contract instance to communicate with the smart contract, whether the user is logged in or not and the user's level. The second custom hook is called *useDataCenter* and it is responsible for getting and storing data related to the smart contract. It depends on the data from the *useUserManagement* hook, which it requires to be successfully initialized. Both hooks are initialized in the *App.js* component and made available to all child components through the application's context. All the components are now able to get their data from the same source. Inside the custom hooks, I also added listeners for events emitted by the smart contract, so the user interface can then refresh itself with the updated data.

To make the code of the hooks and the components cleaner and easier to understand, I extracted parts of the code into individual files. This way components can also share code, reducing the overall code size.

Since I was already familiar with react, I didn't encounter any major problems, except that I often had to search the ethers API documentation, because I used ethers for the first time.

4.4.4 Voting App

I wrote the voting app with react and created the project using Create-React-App. Like in the admin app, the most important packages I used besides react, were ethers for communication with the smart contract, react router for client-side routing and reactstrap for bootstrapping the user interface of the app.

Unlike in the admin app where the admins need to install Metamask to log in to the app, the voters can directly login using the poll id and the mnemonic. This is because it is impractical to require every voter to install Metamask and it also can't be expected from every voter to correctly install Metamask. Also, the voters don't need to access the application as often as the admins, which is why it doesn't make sense for the voters to use an account management tool.

I first started working on the login component and logic and then moved on to the vote casting page and verify vote page. Finally, I also wrote a small component that allows the voters to scan the QR codes on their access letters to log in, so they don't have to enter their credentials manually. Like in the admin app I focused on making the voting app work at its core in the first version, leaving away all less important features.

When I came back to the voting app to make some improvements, I decided to create a custom hook like in the admin app. Unlike in the admin app, I only created one hook instead of two hooks, because the voting app needs to store fewer data and do less communication with the contract than the admin app. Most of the application's code that communicates with the smart contract, like logging in or getting the poll details, is in this custom hook called *useVoteCenter*. The hook also stores most of the application's state concerning data related to the poll or the voter. The instance of the *useVoteCenter* hook is in the *App.js* component and made available

[49] https://reactjs.org/docs/hooks-intro.html

to all child components through the application's context. Code that does tasks like generating the meta transaction and getting the poll, were extracted from the hook to multiple separate files in a *utils* folder. This makes the code inside the hook cleaner and easier to understand.

Because I used the same technologies as in the admin app which I wrote before the voting app, I didn't encounter a lot of major problems. The only major problem I ran into was that the QR code scanning feature at first didn't work on smartphones. The problem was that mobile browsers require a different content security policy to access the camera than desktop browsers. To fix the problem, I needed to change the content security policy of the server that hosts the voting app, not the code of the voting app itself.

4.4.5 CLI Tool

The reason why I didn't make the CLI tool a web application, was because the application needs to be able to run offline for security reasons (RSA key generation and Ethereum account generation should be done offline), which makes it impractical to use a web app. The reason why I then didn't make it a desktop app was because it would take a lot more time than a simple CLI tool and I didn't know if I had enough time left to build a desktop app.

After having decided to make it a CLI tool, I searched for JavaScript frameworks that make it easier to build CLI tools and found one called *INK*[50]. I decided to use INK because it allows me to use react to build interactive command-line tools.

I first created a menu that allows the user to choose between the different actions of the CLI tool and then proceeded to write the different components and functions of the tool. I usually tried to include the functionality inside the different app components, as long as it wasn't too complex. But if this was the case, I extracted the code to a separate JavaScript file which I then imported inside the component.

When I was writing the first version of the CLI tool, I left the feature away for later that verifies the verification hashes and their signatures by comparing it to the data stored on the blockchain because the code in the smart contract that is responsible for verifying and storing the verification hashes wasn't finished yet. After I finished the responsible code on the smart contract, I came back to the CLI tool to add the code that compares the verification hashes and the registrars recovered from the signatures with the smart contract.

4.4.6 Relay Server

The relay server is written in JavaScript and runs on NodeJS. I used the ethers package to communicate with the smart contract and express to run the webserver.

The relay server wasn't a part of the e-voting system when I initially designed it. The reason why the relay server is necessary is that the voters' transactions for casting their votes can't be paid by the voter accounts themselves.

[50] https://github.com/vadimdemedes/ink

First, I determined the structure of the meta transactions for my project, which looks like this:

```
const metaTransaction = {
    transaction: {
        contractAddress, // the address of the eVote smart contract
        pollID, // the id of the poll that is voted on
        voteHash, // the hash of the vote
        encryptedVote, // the encrypted vote
        voter, // the voter's address
    },
    signature // the voter's signature of the transaction’s hash
}
```

Listing 4 Structure of a meta transaction

The meta transaction is generated in the voting app and the relay then receives the meta transaction via an HTTPS request. The code inside the relay server is very simple. The relay first extracts the details of the meta transaction from the HTTPS request’s body. Then the relay verifies if the transaction is meant for the same smart contract it is programmed to relay transactions to. To verify if the meta transaction is correct, it uses ethers’ *estimateGas* function, which runs the vote function of the smart contract with the contents from the meta transaction as parameters on one of the web3 provider’s Ethereum nodes to estimate the gas required, but without actually making a transaction. If the gas required can successfully be estimated, this means that the transaction will not revert, and everything should be correct with the meta transaction. The relay then makes the actual transaction, which requires the payment of a transaction fee. To make the transaction and to pay for the fees, the relay uses an already pre-funded Ethereum account.

After writing the relay, I needed to change the code of the smart contract and the voting app to work with the meta transactions.

The only function on the smart contract that should work with meta transactions is the *vote* function, which casts the voters’ votes. To make the function work with meta transactions, only a few more parameters and a few lines of code inside the function needed to be added, which recreate the transaction hash from the parameters and recover the signer address from the signature.

The changes that were necessary to make the voting app compatible with meta transactions were a bit more complex than the changes necessary in the smart contract. The code that directly sends the transaction to the smart contract needed to be removed. In its place, I added code that can generate a meta transaction, which requires the generation of a transaction hash from the transaction’s parameters and afterwards the creation of a signature for the transaction hash. The meta transaction then is sent in JSON format in an HTTPS request to the relay. Finally, the voting app listens for the contract’s *EncryptedVote* event containing the details of the meta transaction that were relayed. When it receives the event, the voting app knows that the vote has been cast successfully.

4.4.7 Final Implementation

After having finished the frontends and the relay server, I wrote an express server to host the builds of the admin and voting app, and a documentation on how to make an election with the e-voting system. I also integrated the relay server into this express server. The server also takes care of the content security policies of the frontends. I also created a docker file for this server project so it can be run on Kubernetes[52].

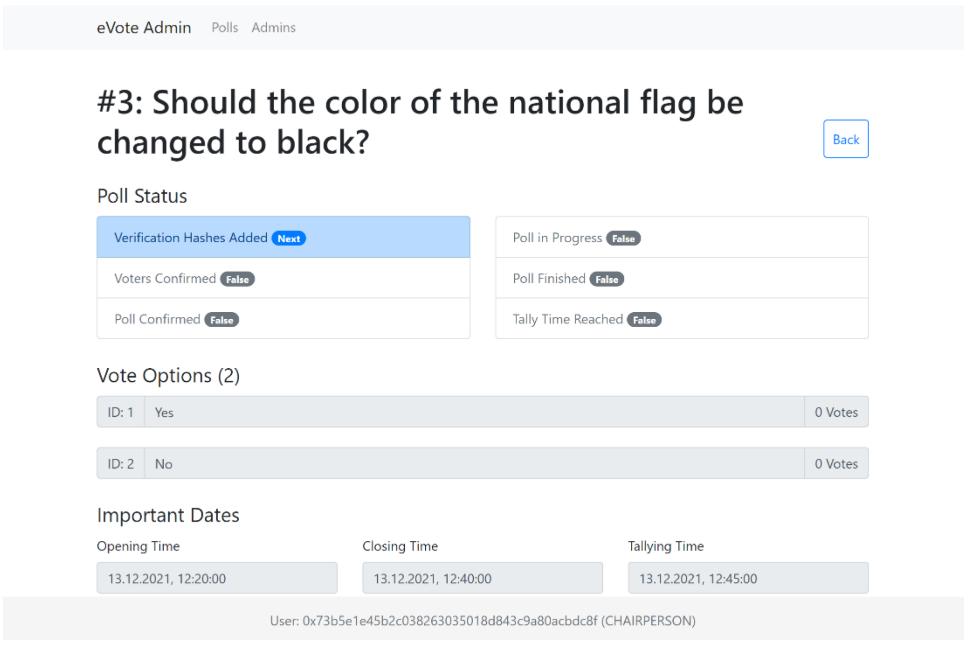
When I finished writing all the software components, I put them all together into one big project called *evotesystem*. Then I added some scripts to this project to automate some of the repetitive tasks during the setup and build process, that needed to be done manually before, like distributing configuration files, moving the builds from the admin and voting app to the server or installing npm packages in all the projects. This made it a lot easier and faster to have an instance of the e-voting system fully running for testing.

5 Product

This chapter focuses on the final product. A more detailed and technical overview of the admin levels, software components and the most important steps will be given. Additionally, the operating costs of the voting system that arise because of transaction fees will be calculated and compared between the main Ethereum network and Polygon, an Ethereum based blockchain. It will also be analysed whether the requirements that have been set out at the beginning of the paper in the Requirements chapter have been reached or not. Finally, there will also be shown possible improvements that could be made in a future version of the voting system.

5.1 Pictures

To show how the different apps look like, this chapter shows some pictures of the final voting app, admin app, registration app and CLI tool.



#3: Should the color of the national flag be changed to black?

Poll Status

Verification Hashes Added Next
Voters Confirmed False
Poll Confirmed False
Poll in Progress False
Poll Finished False
Tally Time Reached False

Vote Options (2)

ID: 1	Yes	0 Votes
ID: 2	No	0 Votes

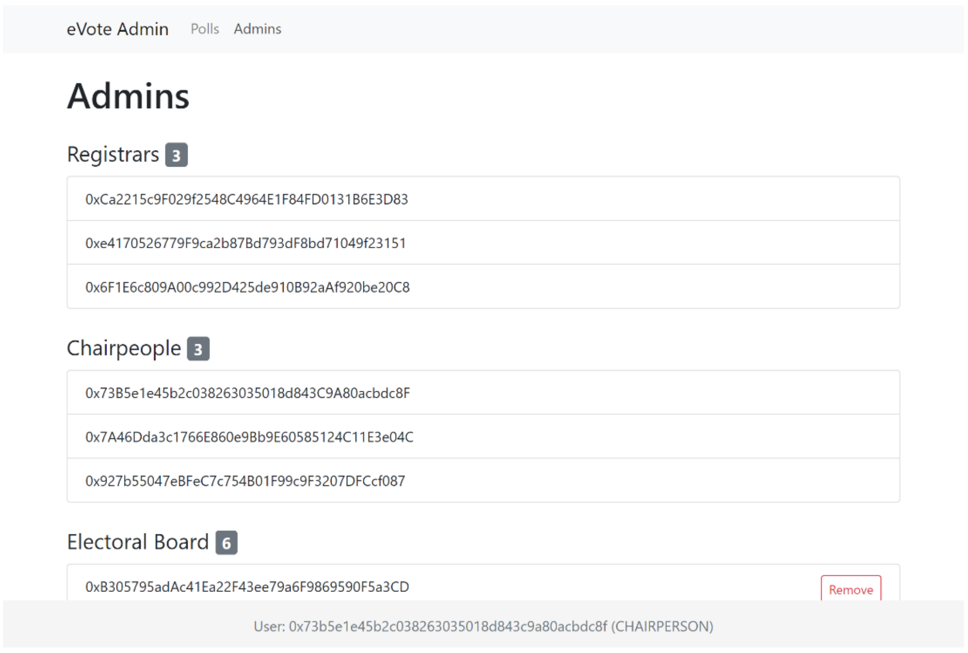
Important Dates

Opening Time	Closing Time	Tallying Time
13.12.2021, 12:20:00	13.12.2021, 12:40:00	13.12.2021, 12:45:00

User: 0x73b5e1e45b2c038263035018d843c9a80acbdc8f (CHAIRPERSON)

Figure 12 Single poll page of the admin app

[52] https://kubernetes.io


eVote Admin Polls Admins

Admins

Registrars 3

0xCa2215c9F029f2548C4964E1F84FD0131B6E3D83

0xe4170526779F9ca2b87Bd793dF8bd71049f23151

0x6F1E6c809A00c992D425de910B92aAf920be20C8

Chairpeople 3

0x73B5e1e45b2c038263035018d843C9A80acbdc8F

0x7A46Dda3c1766E860e9Bb9E60585124C11E3e04C

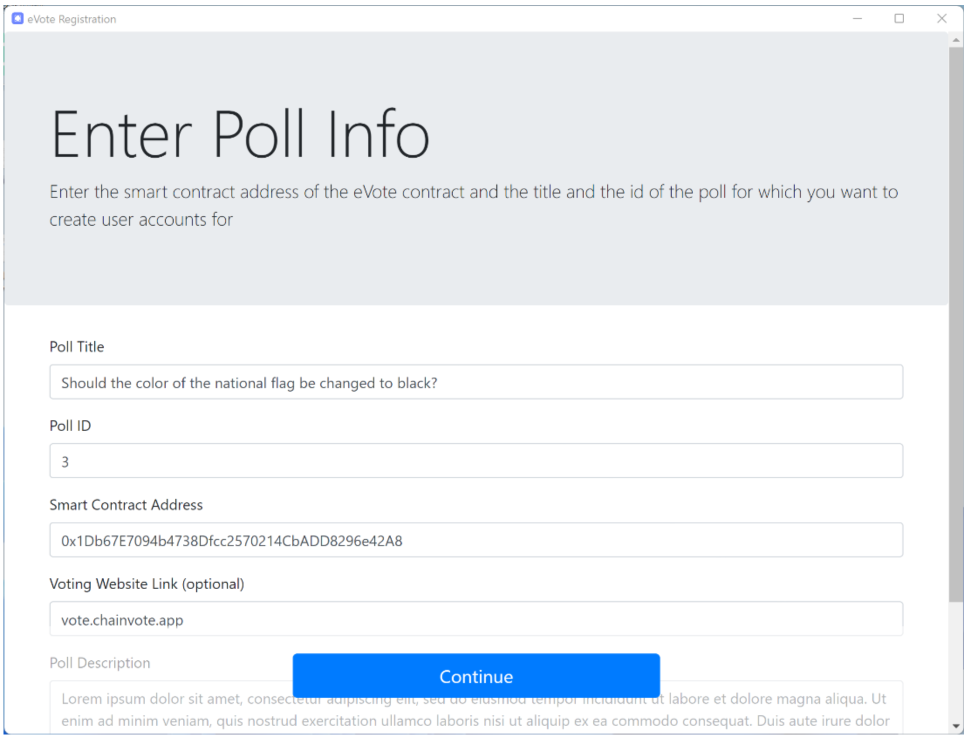
0x927b55047eBFeC7c754B01F99c9F3207DFCcf087

Electoral Board 6

0xB305795adAc41Ea22F43ee79a6F9869590F5a3CD Remove

User: 0x73b5e1e45b2c038263035018d843c9a80acbdc8f (CHAIRPERSON)

Figure 13 Admins page of the admin app


eVote Registration

Enter Poll Info

Enter the smart contract address of the eVote contract and the title and the id of the poll for which you want to create user accounts for

Poll Title

Should the color of the national flag be changed to black?

Poll ID

3

Smart Contract Address

0x1Db67E7094b4738Dfcc2570214CbADD8296e42A8

Voting Website Link (optional)

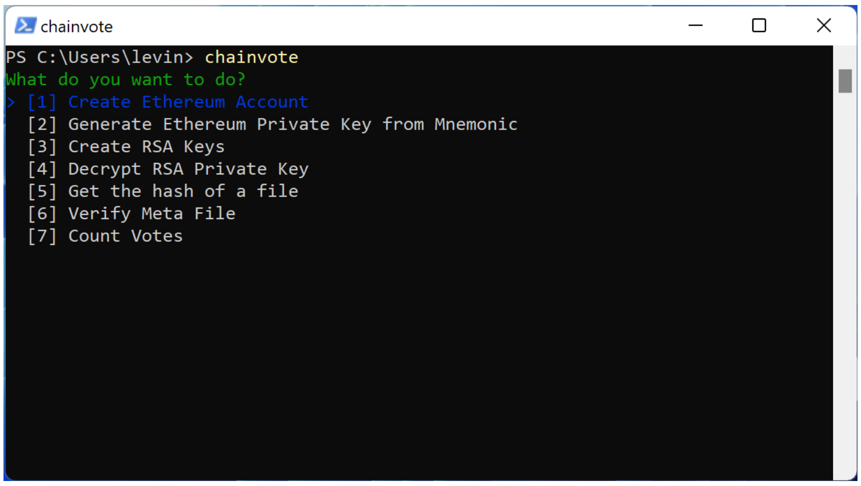
vote.chainvote.app

Poll Description

Continue

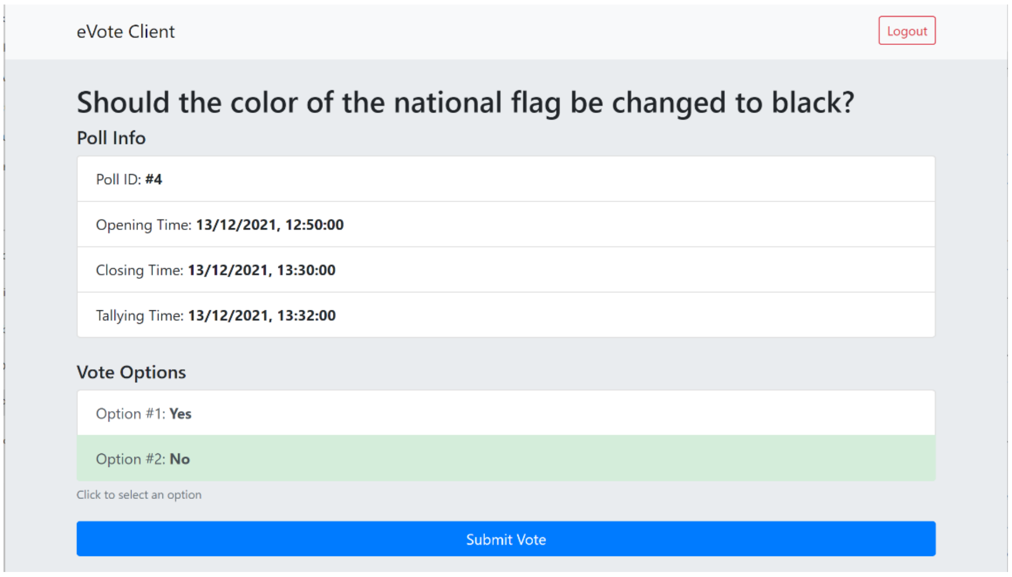
Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor

Figure 14 Poll info page of the registration app



```
PS C:\Users\levin> chainvote
What do you want to do?
> [1] Create Ethereum Account
  [2] Generate Ethereum Private Key from Mnemonic
  [3] Create RSA Keys
  [4] Decrypt RSA Private Key
  [5] Get the hash of a file
  [6] Verify Meta File
  [7] Count Votes
```

Figure 15 Menu of the CLI tool



eVote Client

Should the color of the national flag be changed to black?

Poll Info

Poll ID: **#4**

Opening Time: **13/12/2021, 12:50:00**

Closing Time: **13/12/2021, 13:30:00**

Tallying Time: **13/12/2021, 13:32:00**

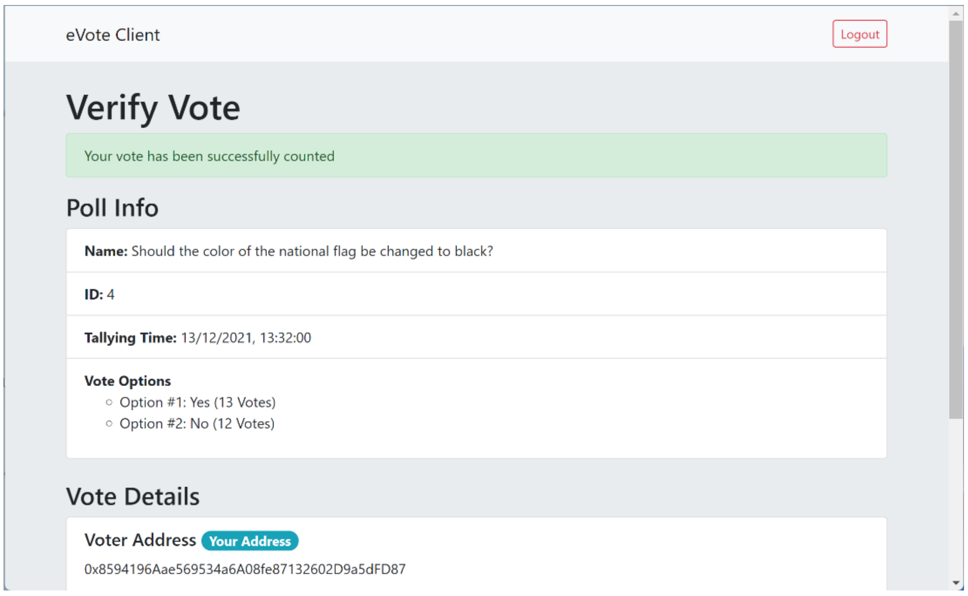
Vote Options

Option #1: **Yes**

Option #2: **No**

Click to select an option

Figure 16 Voter logged in to the voting app has "No" as option selected before submitting their vote



eVote Client

Verify Vote

Your vote has been successfully counted

Poll Info

Name: Should the color of the national flag be changed to black?

ID: 4

Tallying Time: 13/12/2021, 13:32:00

Vote Options
- Option #1: Yes (13 Votes)
- Option #2: No (12 Votes)

Vote Details

Voter Address Your Address

0x8594196Aae569534a6A08fe87132602D9a5dFD87

Figure 17 The verify vote page of the voting app after the tallying of the votes

5.2 Technical Overview

This chapter gives an overview of the voting system. It shows and describes all the software components that are required and the different admin levels and their areas of responsibility. Important sequences like the process of creating a new poll or user voting will also be shown and explained.

5.2.1 Admin Levels

The voting system has three different types of admin roles that all have different capabilities when it comes to holding an election. The reason why there are three different roles is to divide power between the different groups to provide for checks between the groups and prevent the concentration of unchecked power.

5.2.1.1 Registrars

There are 3 registrars, whose duties are to create voter accounts in the registration app and to add the verification hashes to the smart contract. They can also vote on and start add and removal requests for electoral board members along with the chairpeople.

5.2.1.2 Chairpeople

There are 3 chairpeople, whose duties are to create new polls and to add, verify and confirm voters. They can also vote on and start add and removal requests for electoral board members along with the registrars.

5.2.1.3 Electoral Board

There are 6 electoral board members, whose duty at the beginning of an election is to create an RSA key pair that will be used to encrypt and decrypt votes. After a poll has been created and voters confirmed, they have to verify the voters of a poll, make sure that everything is correct with the poll and then confirm the poll. At the end of the poll, they are responsible for decrypting the RSA private key and counting the votes. They can also vote on and start add and removal requests for registrars and chairpeople.

5.2.2 Software Components

This section gives an overview of the different software components of the e-voting system and describes what they do.

5.2.2.1 Smart Contract

The smart contract is the core component of the voting system, as it contains the voting logic and stores all the data. The data that it stores in its state is the polls and their details, how many votes each vote option received, the current admins, the registered voters for each poll and open two-thirds requests. The encrypted votes are stored in the contract's event logs. The smart contract and the data stored on it can be viewed by anyone. The votes on the contract are, however, encrypted until the respective poll reaches tallying time. The admins and the voters don't access the smart contract directly, all the communication happens over the admin app, voting app, relay or the CLI tool, which use the contract's interface to communicate with it. The smart contract is written in solidity.

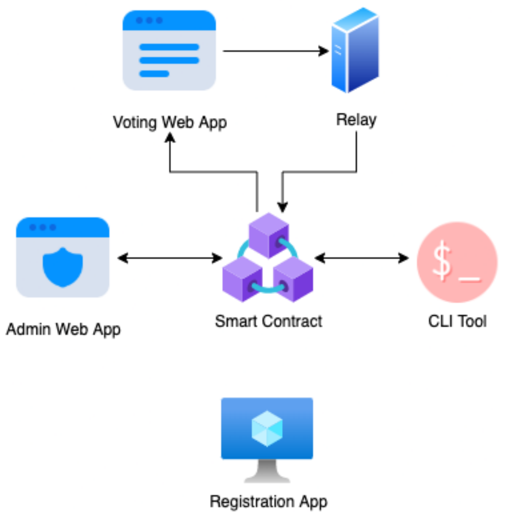


Figure 18 Overview of the voting system's software components

5.2.2.2 Admin App

The admin web app allows admins to do administrative tasks by showing them important information from the smart contract and allowing them to call functions that are restricted to the admins. It allows the chairpeople to create new polls, add and remove voters and confirm voters. The registrars can add verification hashes that have been generated in the registration app and the electoral board can confirm polls to allow voters to vote on them. The admins can also add and remove admins of the contract by creating add or removal requests for admins. The app also displays the admins a list of all polls that have been created and the details of those polls, like the verification hashes, public RSA key, registered voters, cast votes and more. The admin app is written in HTML, CSS and JavaScript, using the react framework and the help of other npm packages, the most important ones being ethers for communication with the smart contract and reactstrap for bootstrapping the UI.

5.2.2.3 Voting App

The main functionality of the voting web app is to allow voters to choose a vote option and cast their vote. Voters can log into it using the mnemonics they received on their access letters, either by manually entering them or scanning the QR Codes on the access letters. After having cast their vote, the voter can also download a receipt. The details of the vote and other details like the block number and the transaction hash can be viewed immediately after the voter cast their vote or anytime later when the voter logs in again with their credentials. Voters can also check if their vote has already been counted and the app also allows them to verify their vote receipt. Like the admin app, the voting app is also written in HTML, CSS and JavaScript. It also uses the react framework and the ethers and reactstrap packages.

5.2.2.4 Registration App

The registration app is a desktop application that allows registrars to import a list of users in a CSV file which should receive access to vote on a specific poll and the app then creates Ethereum accounts for those users. The app then also generates access letters that contain the mnemonic of the account along with the details of the poll and some additional information. The Ethereum addresses of the voters' accounts are saved in a separate file, so they can later be registered as voters on the smart contract. The app additionally generates a file with the names and the physical addresses of the voters. In the end, the app hashes all the outputs and creates a hash tree with them which then is signed by the registrars' private keys. The hash tree and the signatures are added to all the output files in a file called *meta.json*. The registration app is written in JavaScript using electron. The user interface is made with HTML, CSS and JavaScript using the react framework and reactstrap among other packages.

5.2.2.5 CLI Tool

The CLI tool allows admins to do multiple things that are necessary to carry out an election. These things include:

- Creating RSA keys which are necessary to encrypt and decrypt the votes that are cast
- Decrypting RSA private keys
- Verifying a meta file
- Counting votes

Additionally, the tool also includes some extra functions which are supposed to help the admins, but could also be done with external tools:

- Getting the hash of a file

- Creating Ethereum accounts (mnemonic, private key and address) and deriving private keys and addresses from mnemonics

The CLI tool is written in JavaScript using INK.

5.2.2.6 Relay

The relay forwards the votes of the voters to the smart contract and pays the transaction fee that arises when casting a vote, so the voters don't have to bear these costs. It is written in JavaScript and integrated into the express server that hosts the voting and the admin app.

5.2.3 Initial Contract Deployment

Before a poll can be created on the smart contract or any action executed on the smart contract, it first needs to be deployed. After the contract has been deployed, the frontends can also be deployed and the CLI tool set up, which then can interact with the smart contract. To deploy a new instance of the smart contract, multiple steps need to be taken.

Step 1: Determining the initial admins

The poll organizer determines the people that will be the initial admins of the smart contract.

Step 2: Admins generate Ethereum accounts

The people that have been appointed as admins now use the CLI tool's *Generate Ethereum Account* function to generate an Ethereum account for themselves. The CLI tool will display the admin their mnemonic, Ethereum private key and address, which they need to write down and store somewhere safe.

Step 3: Technical staff configures admin file

The admins then hand over their address to the technical staff, so they can place the addresses of the admins in the admin configuration file of the *evotesystem* project, which contains the addresses that will be used as admins when the contract is deployed.

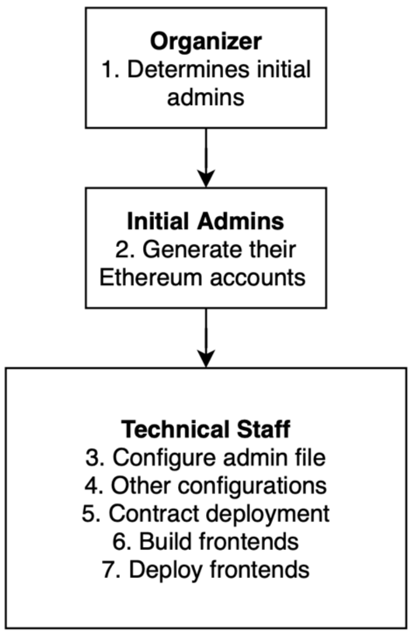


Figure 19 System deployment steps

Step 4: Technical staff configures the rest of the project

The technical staff configures the other files that are necessary for deployment. This configuration includes the URL of the node provider that should be used, the name of the blockchain the smart contract should be deployed to, which account should be used for contract deployment and which Ethereum account should be used in the relay to pay the transaction fees. Additionally, there are some other optional configurations the technical staff can make.

Step 5: Technical staff deploys contract

After having finished the configuration process, the technical staff deploys the contract to the desired blockchain.

Step 6: Technical staff builds frontends

After having deployed the contract, the technical staff add the contract's address to the contract configuration file. Then they use the CLI tool to distribute the configuration file to the frontends and to build the frontends.

Step 7: Technical staff deploys frontend.

After having built the frontends, the technical staff deploys the server. The server hosts the relay, the voting app, the admin app, and, in development, a documentation for the project.

After the deployment of the server, the *evotesystem* is ready to use.

5.2.4 Creating a Poll

This section explains in detail the steps that happen when the admins create a new poll. It is assumed that at this point the organizers of the election already decided what the poll is going to be about, what the possible vote options are, and which voters should be eligible to vote on the poll.

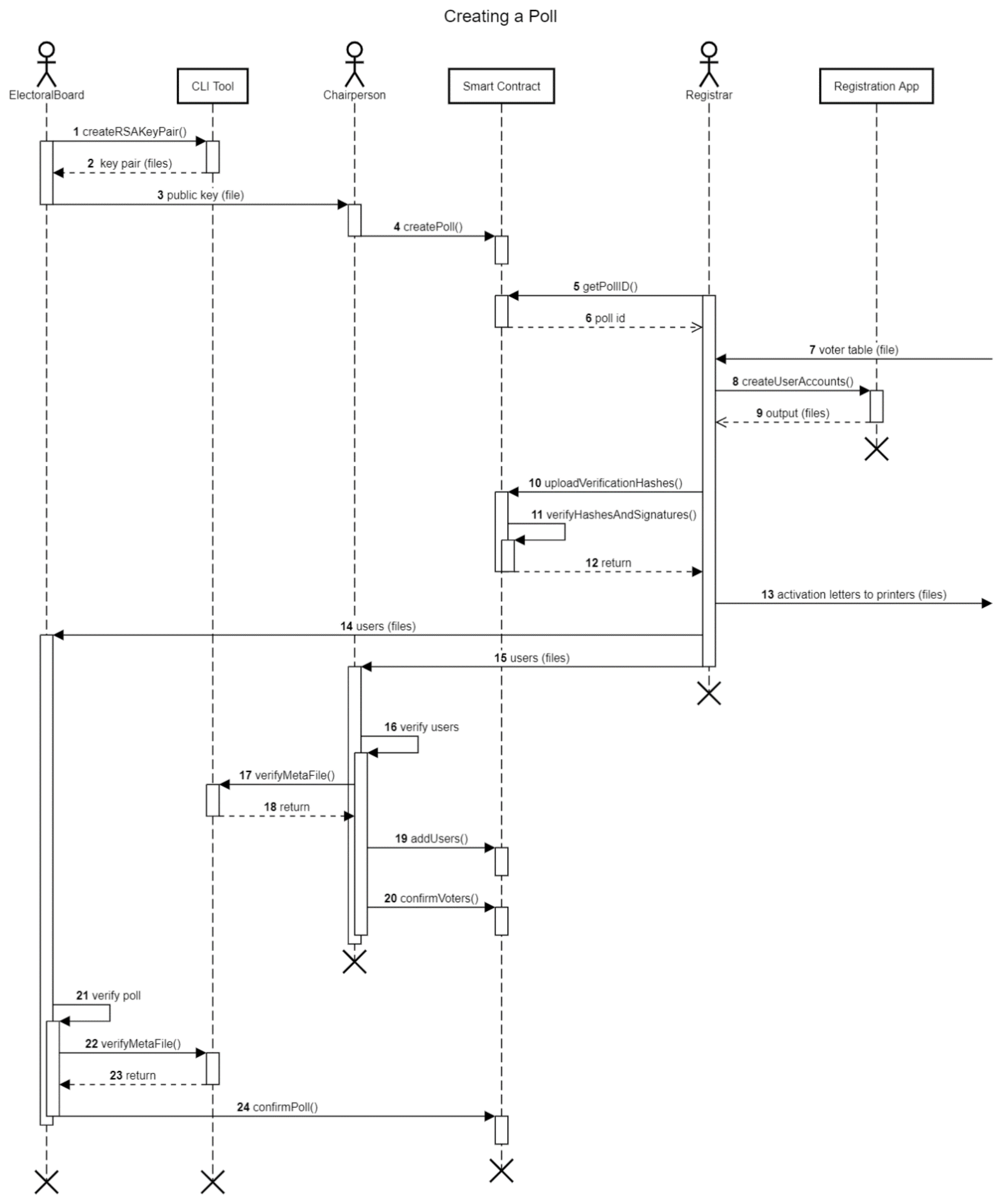


Figure 20 Sequence diagram of the poll creation process

Steps 1 and 2: Electoral board creates RSA key pair

To ensure that the votes stay secret until the poll reaches its tally time, the votes need to be encrypted by an RSA public key. The system uses a 4096-bit RSA key pair. To generate the key pair, the electoral board meets and use the CLI tool installed on an offline computer and choose its *Create RSA Keys* option. The program will ask where the output should be saved to and then prompts the electoral board members to enter their Ethereum private keys after another. No one other than the electoral board member to whom the key belongs must see the key.

The admins' private keys are used to generate a passphrase which is used to encrypt the RSA private key. This ensures that the electoral board members can only decrypt the private key and decrypt and count the votes when all of them enter their Ethereum private key. This should prevent a rogue electoral board member from encrypting the votes before the election ends.

After the program created the RSA key pair, it saves the public key and the encrypted private key in *pem* format to the specified location along with a file called *positions.json*. This file contains information about how the passphrase for the RSA private key was generated from the Ethereum private keys of the electoral board members and it is necessary to decrypt the private key at the end of the election. Each electoral board member keeps a copy of the encrypted RSA key and the *positions.json* file, which they must keep until the election reaches tallying time and the votes have been counted.

Step 3: Electoral board hands over the public key to the chairpeople

After the electoral board generated the RSA keys, they hand the RSA public key over to the chairpeople.

Step 4: Chairperson creates a new poll

One chairperson now creates a new poll with the poll name, vote options, opening, closing and tallying times determined by the poll organizer and the public key received from the electoral board. The chairperson can also specify a content hash, which can be a hash of the poll description or a video explaining what the poll is about and allows the voters to be sure that the description or video is genuine.

Steps 5 to 9: Registrars create voter accounts

After the poll has been created, the registrars can create voter accounts for the poll in the registration app. For security reasons the computer that is used to generate the voter accounts must be offline. The registration app requires the registrars to enter the contract address, the poll id and the poll name before creating the voter accounts. Additionally, the registrars can also enter a poll description and a voting link, the URL of the website where voters can vote, which will be printed on the activation letters. After entering the details of the poll, the registrars need to select a CSV file, which contains the details of the voters that should be registered. This file should be provided to all the admins by the poll organizers. The registration app then generates a 12 word long mnemonic for each user. The mnemonic then gets split into two, the first half will be on the first activation letter and the second half on the second activation letter.

Besides the mnemonics, the activation letters also contain the poll id and the name of the poll, which are also contained in a QR code on the activation letters. The letters also contain the voter's address and name and if specified a poll description and voting link.

After the mnemonics and activation letters have been generated, the registrars need to enter their Ethereum private keys to sign the output and prove its genuinity. The way how the output is hashed and signed to create verification hashes and how the verification hashes work is explained in more detail in the Verification Hashes chapter. After the output has been signed, the registrars can download the different output files.

Steps 10 to 12: Registrars upload verification hashes

One of the registrars uploads the verification hashes to the smart contract. The smart contract then checks if the hash tree is correct and if the signatures of the registrars are correct.

Step 13: Activation letters to printers

After the verification hashes have been added, the activation letters are sent to the printers. The printers will only start printing the activation letters once the poll has been confirmed.

Steps 14 and 15: Users to chairpeople and electoral board

The voter details and the voters' Ethereum addresses are then given to the chairpeople and the electoral board.

Step 16 to 18: Chairpeople verify voters

When the registrars receive the voter details and their Ethereum addresses, they verify that the data they received is correct, by doing the following:

- Making sure that the correct poll id, contract address and poll name are included in the *users.json* and *addresses.json* files.
- Making sure the correct voters have been registered (i.e. making sure the *users.json* contain the same voters as the CSV file with the voters that was given to the admins by the poll organizers)
- Hashing the files, which can be done with the CLI tool, and comparing the hashes to the meta file.
- Using the CLI tool's *Verify Meta File* function, which verifies that the meta files included with the users and Ethereum addresses are valid. The program does this by checking if the hash tree is valid, checking the hashes match the ones on the smart contract which were previously added by the registrars and by verifying the signatures.

Steps 19 and 20: Chairpeople add and confirm voters

If the voter details and the Ethereum addresses the chairpeople received are correct, one of the chairpeople adds the voters to the poll. Then the chairpeople open a two-thirds request for voter confirmation, which needs to be approved by two out of three chairpeople voting on it. After confirming the voters, the poll status will be updated to *voters confirmed*.

Steps 21 to 23: Electoral board verifies voters and poll

This is the final review of the poll before it gets confirmed. The electoral board does the same verification steps that have been done in steps 16 to 18 by the chairpeople. Additionally, they verify that everything else with the poll is correct: The poll name, the vote options, the opening/closing/tallying times and very important, the RSA public key.

Step 24: Electoral board confirms poll

If the electoral board finds everything to be correct, they open a poll confirmation request, which requires two-thirds of the electoral board members' votes to pass. After 4 out of the 6 electoral board members voted on it, the poll is confirmed, and voters can start voting on the poll as soon as the opening time is reached.

5.2.5 User Voting

This chapter explains in detail the steps that happen when a voter votes on a poll. It is assumed that at this point the poll has already been successfully created with the voters registered, the poll is open (opening time exceeded but closing time not surpassed) and the voter received their access letters and has the voting website open.

Step 1: Voter enters credentials

The voter enters their credentials, the first and the second half of the mnemonic along with the poll id, which are all stated on the access letters. The voter can enter their credentials either by entering them manually in a form or by scanning the two QR codes on the access letters, which also contain the credentials. After entering the credentials, the user clicks the login button.

Steps 2 and 3: Voting app checks if the voter is eligible to vote

After receiving the credentials, the voting app first checks if the mnemonic the voter entered is valid. After that, the application creates a provider and a wallet from the mnemonic, which are used to create a contract instance. The app then calls the contract's *viewVoterStatus()* method, which returns either 0, meaning that the user isn't registered to vote, 1, meaning that the user is registered and hasn't voted yet, 2, meaning that the user is registered to vote but has already voted, or an error, in the case, that something went wrong, which is usually the case when the poll entered by the user doesn't exist.

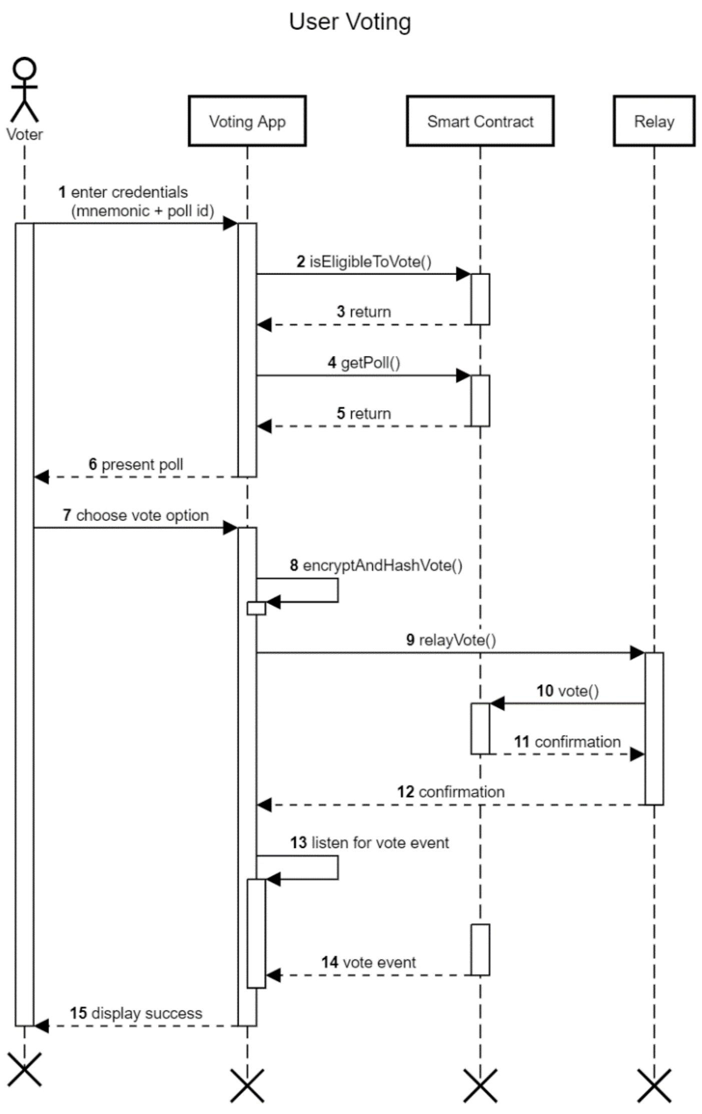


Figure 21 Sequence diagram of the user voting process

If the *viewVoterStatus()* function returns 0 or an error, the login page is displayed again, but with an error message.

If the function returns 2 (voter already voted), the voter will be redirected to the *verify vote* page, where the voter can view the transaction details of their vote and can verify their vote.

When the user is eligible to vote and 1 is returned, the application will get the poll details from the smart contract and display the voting page.

Steps 4 to 6: Getting the poll details

When the voter is eligible to vote, the voting app will first get the poll details with the poll id the user entered at login, using the contract's *idToPoll()* function. Most of the information required is returned by this function, most importantly the poll title, the public RSA key used to encrypt the voter's vote later, the opening/closing/tallying dates and times along with the vote options count. This function, however, does not return the vote options themselves, only how many of them there are. With the vote options count and the poll id, the contract's *viewVoteOption()* function can be used to get the poll's vote options. After the voting app receives the information, it will display them on the voting page to the voter.

Step 7: Voter chooses an option to vote for

The voter now takes their time and chooses an option to vote for. When the voter decided, they click the submit vote button and a modal dialogue will ask them to confirm their choice.

Step 8: Generating the vote

When the voter clicks the confirm button on the modal, the vote is generated. After the vote is generated (salt, hashed vote, and encrypted vote are generated), the application uses the details of the vote to generate a meta transaction.

Step 9: Relaying the vote

The meta transaction generated in the previous step is now sent to a relay. The relay verifies if the meta transaction contains all the required information if the signature is correct and if the voter is eligible to vote by test running the *vote()* function of the smart contract with the parameters of the meta transaction on a node.

Steps 10 to 12: Casting the vote

If the transaction is valid, the relay uses the contract's *vote()* function to cast the vote. To pay for the transaction fee that arises when casting the vote, the relay will use an Ethereum account that it was preconfigured with and has enough funds to pay for it. After the relay sent the transaction to the blockchain, it receives a confirmation, which it will pass on to the voting app.

Steps 13 and 14: Listening for vote event

When the contract receives the confirmation from the relay that the transaction has been submitted to the blockchain, it starts listening for the contract's *EncryptedVote* events, which are emitted once a vote has been received by the smart contract. When the voting app receives an *EncryptedVote* event, containing the voter's address, the poll's id, the vote hash and the encrypted vote that was generated in step 8, it knows that the vote has been cast successfully.

Step 15: Displaying the success

After receiving the *EncryptedVote* event, the voting app will display a success message and other important information regarding the vote to the voter. This information includes the option the voter voted for, the hash of the vote, the UUID (salt) that was used to generate the vote and the Ethereum address of the relay that was used to cast the vote. The app also allows the voter to download all the details of their vote in a *chainvote* file, so they can completely verify that their vote has been counted at the end of the poll. The voter can also go to the verify page, where they can see more details regarding their vote like the transaction details. There they can also verify their *chainvote* file and view if their vote has already been counted.

5.2.6 Counting Votes

After the tally time of a poll has been reached the electoral board will decrypt the RSA private key to count the votes.

Step 1: Decrypting the RSA private key

The electoral board members use the CLI tool's *Decrypt RSA Private Key* function to decrypt the RSA private key. They enter the file location of the encrypted RSA key and the positions file. Then every electoral board member needs to enter their Ethereum private key, which is necessary to generate the passphrase to decrypt the RSA private key. After encrypting the RSA private key, it gets saved to the computer.

Step 2: Counting the votes

To count the votes of a poll, the electoral board needs to use the CLI tool's *Count Votes* function. First, they need to enter the private key of an Ethereum account that has enough funds on it to pay for the gas costs that arise when counting the votes. Then they need to specify the id of the poll, whose votes should be counted. Finally, they need to specify the path of the decrypted RSA private key.

The program will then get all the encrypted votes from the smart contract's event log and will display an overview to the electoral board, showing how many votes have already been counted, how many votes haven't been counted and how many votes have an error. Then the electoral board can proceed to decrypt the votes by hitting enter. After decrypting the votes, the program will show how many votes it was able to decrypt. Finally, the program submits the votes to the smart contract for counting.

5.3 Operating Costs

As mentioned in the Gas and Transaction Fees chapter, it costs money to make transactions on a blockchain to call a function on a smart contract. These costs are called gas costs or transaction fees and they depend on the blockchain used, the number of transactions currently being made on the entire blockchain and the price of the cryptocurrency used to pay the transaction fee.

The cost associated with running an election is an important consideration when choosing a blockchain to deploy the contract to as a developer or as an election organizers when deciding whether it is feasible to run an election on a blockchain instead of a traditional election, where the votes are cast via mail, or a centralized e-voting system.

To find out how much it would cost to run an entire election with the *evotesystem*, I calculated the total amount of transaction fees that occur when running an election with 1000 voters from start to beginning. I calculated the costs for two different blockchains. For the main Ethereum blockchain and for Polygon, an Ethereum based proof of stake[59] blockchain. Because gas prices are always fluctuating, averages from the last 3 months[60] were used.

Action	*Gas Required*
Contract Deployment	5025212
Poll Creation	1038406
Adding Verification Hashes	237723
Adding Voters	27504600
Confirming Voters	258984
Confirming Poll	402502
Casting Votes	86869000
Counting Votes	13667880
Total Units of Gas	**135004307**

Table 3 Units of gas required to conduct an election

	Ethereum	**Polygon Proof of Stake**
Average Gas Price per Unit (GWEI)	116.961[61]	74.17152247[62]
Total Gas Cost in GWEI	15790238751	10013474990
Total Gas Cost in the Blockchain's Currency	15.79023875 ETH	10.01347499 MATIC
Price of the Blockchain's currency[63] (CHF)	3551.52 CHF	1.75 CHF
Total Gas Cost (CHF)	**56079.35 CHF**	**17.52 CHF**

Table 4 Gas prices on the different blockchains

In Table 3, you can see the amount of gas that is required for every step of the election process and in Table 4, you can see the gas costs for the two blockchains.

The total gas cost in GWEI[64] is calculated by multiplying the total units of gas times the average gas price per unit in GWEI. The total gas cost in the blockchain's currency is calculated by dividing the total gas costs in GWEI by one billion. The gas fee on Ethereum is paid in Ether, and the gas fee on Polygon is paid in Matic. To get the total gas cost in Swiss francs, the price of one unit (1 Ether / 1 Matic) of the blockchain's currency in Swiss francs is multiplied by the total gas cost in the blockchain's currency.

As you can see in Table 4, the cost of running an election on Polygon is much lower than on the Ethereum main net, where it would probably be too expensive for most organizers to run an election. On Polygon the transaction fees are much lower, which makes it financially a lot more appealing than Ethereum. However, the cheaper price of Polygon also comes at a loss of decentralization and security, which makes it less ideal for governmental elections. There are a lot of other Ethereum based blockchains, like Arbitrum, Optimism and Avalanche. Some of them cost more and are more secure, others are cheaper but less secure. It won't be analyzed which

[59] https://en.wikipedia.org/wiki/Proof_of_stake
[60] 13.9.2021 - 12.12.2021
[61] https://etherscan.io/chart/gasprice (Accessed: 13.12.2021)
[62] https://polygonscan.com/chart/gasprice (Accessed: 13.12.2021)
[63] Prices on the 13.12.2021 from https://coinbase.com
[64] 1 GWEI is one unit of the cryptocurrency divided by a billion. It is used to specify gas prices.

blockchain the smart contract should be deployed to for production because this is out of the scope of this paper.

It is important to note that in addition to the transaction fees, also the costs of hosting the servers and the cost of printing the activation letters need to be considered.

5.4 Requirement Analysis

In the Requirements chapter at the beginning of the paper, the requirements the e-voting system should meet were set out. In this chapter, it is analysed if and to what extent these requirements have been met.

First, we are going to look at the requirements that were taken from the "Anforderungskatalog für eidgenössische Volksabstimmungen mit der elektronischen Stimmabgabe".

Requirement	Status	Implementation
Individual verifiability	Reached	The voting system successfully implements individual verifiability by allowing users to verify their vote at any time in the voting app. For verification, the voting app locally regenerates the vote of the voter and then compares it to the vote stored on the blockchain. Voters can also check if their vote has already been counted or not. Voters can also visit Etherscan to verify that their vote has been received or counted.
Complete verifiability	Reached	Complete verifiability is also successfully implemented in the voting system. It is in the first place given by the verification hashes that ensure that the correct voters have been registered. Then during the voting process, complete verifiability is provided by the blockchain and the smart contract, which would show any manipulation attempts made on the election. At the end of the election, anyone can recount the votes for themselves using the RSA private key and the events containing the votes.
User-friendliness	Mostly reached	This requirement has been mostly met. Voters can easily log in with QR codes and the user interface is mostly straightforward, but there is still room for improvement. An example for this would be to use a different term than mnemonic that makes more sense to the general population, like «Access Key» or «Login Key».
Voters with disabilities	Not reached	This requirement has not been completely met. The voting app can in principle be used with voice control, but for time reasons the app hasn't been optimized for screen readers.
Invalid ballots	Mostly reached	This requirement has been partially met. The voting web app won't allow the voters to cast an invalid ballot. However, if a voter for some reason tried to directly send an invalid vote to the smart contract, the smart contract would accept it, as long as the vote is cast on an open poll, the voter is registered and the message signature is correct. This is because the smart contract can't access the contents of a vote. After all, it only receives the vote hashed and encrypted.
Tallying of the votes	Reached	Treating the results of a poll confidentially between decryption and the publication of the results has been reached. Votes are encrypted until the election ends and can't be accessed by anyone until the electoral board decrypts the RSA private key and counts the votes at the tallying time. The system is designed transparently, and the votes are counted publicly, meaning that the results are publicly available as soon as the votes are

		counted. This means that the votes need to be decrypted and counted at the determined publication time of the results to ensure the confidentiality of the results.

Table 5 Requirement analysis of the federal chancellery's catalogue of requirements

Most of these requirements have been completely or mostly met and the remaining requirements could easily be met with some additional effort.

Now we are going to look at the requirements I set out myself.

Requirement	Status	Implementation
Usable for non-governmental elections	Mostly reached	The system can be used without any problems for most non-governmental elections, where the voters should only be able to vote for one option and where there should be a maximum of five options to choose from. However, for elections that aren't very important and only have a few voters, 12 required admins are probably too much. The admin web app only needs a few lines of modifications to work with less than 12 admins and has even been partially designed to allow a switch to a single admin mode, where the smart contract only has one admin. In the registration app also only a few lines would need to be changed to achieve this. The smart contract and the CLI tool need more changes to work with less or only one admin than the admin web app and the registration app, but the changes could also be made fast and relatively easy.
Important data and logic on the blockchain	Reached	This requirement has also clearly been reached because the votes, verification hashes, admins and voters are all stored on the blockchain. The core logic is also on the blockchain, like the counting and casting of the votes, the confirmation of polls and voters, the adding and removing of admins and the validation of the verification hashes.

Table 6 Requirement analysis of the personal requirements

These requirements have also generally been met.

5.5 Possible Improvements

In this chapter, I am going to highlight things I would change in a future version of the project, regarding the code and the design of the entire voting system

5.5.1 Additional Verification Features for the CLI Tool

The CLI tool currently allows admins to verify the correctness of the meta file, but it can't automatically verify if the voter addresses in the *addresses.json* file have all been correctly registered as voters on the smart contract. This feature along with the option to automatically compare multiple of the registration app's output files' hashes with the hashes from the meta file, so the admins don't need to manually verify if the hashes of the output files match the hashes inside the meta file, should be implemented in a future version of the voting system.

5.5.2 Improve Central Data Storage in the Admin App

The admin app currently uses a custom hook called *useDataCenter* which handles most of the tasks related to getting data from the smart contract, like loading polls, two-thirds requests and listening for events. However, some of the components still fetch their own data, like the *Voters* and *Votes* component, which directly get the voters and the votes from the blockchain and not through the custom hook. This is not a major problem since the data is only used in that component and its children and not across the whole application. But the data is fetched again every time the component is remounted, which wouldn't be the case if the voters and votes were stored in the custom hook, which is why I would have moved the code that stores and gets the data in those components to the *useDataCenter* hook as well if I had had enough time to do so.

5.5.3 Separation of Servers

Currently, the voting app, the admin app and the relay server run on a single server. This design works great on a small scale where only a few users access the e-voting system, and it also makes it easier to quickly test the software and deploy it to a small server for a demo with a few users. However, it is not suitable for a production environment, where a lot of users need to use the services, which would be the case in a national election in Switzerland, where approximately 5.5 million people[65] need to access the services to vote.

That is why in a production environment the admin app, the voting app, and the relay would each be placed in its own express server. Then every server would receive a Docker file, so the servers can be containerized. The voting app and the relay server containers would then be deployed to Kubernetes, which will scale the number of containers up and down, depending on how much traffic is coming in: During an election, the number of containers will be scaled up because more users are voting and thus traffic is higher. Contrary, when there is no election ongoing, the number of containers will be scaled down, because only very few people will access the services. This ensures that services won't be overloaded and it also minimizes costs.

The admin app wouldn't be deployed to Kubernetes because a single instance running of the container is enough because the admins are the only people who need to access it and there are only 12 of them. Unlike the voting app and the relay, which need to be accessible to the voters over the internet, the admin app wouldn't be deployed to publicly accessible servers, but instead on a server on a local network, which can be accessed with a VPN. This makes it more secure because among other reasons it prevents the use of DDOS attacks on the server and makes man in the middle attacks less likely.

Another benefit of the separation and containerisation of the services is, that in case one of the servers crashes for some reason, the other servers won't be affected by it and keep running. If the server in the current setup crashes for some reason, not only one service will be unavailable, but all of them.

5.5.4 Adding a Zero-Knowledge Proof and reusable Accounts

When a user votes on a poll, the encrypted vote along with some other information including the user's address is sent to a relay, which then forwards the user's vote to the smart contract, where the vote will be stored, and the user's account status updated to *voted*. This means that a user's Ethereum address and vote are publicly linked together and can be looked up by anyone who has access to the blockchain. After the votes have been decrypted and counted, anyone can look up which Ethereum address voted for what vote option. This prevents the reuse of addresses and access letters across multiple polls because it would allow the profiling of the voters.

Here comes the zero-knowledge proof into play. A zero-knowledge proof is a cryptographic method that allows one party to prove to another party that a given statement is true, without revealing any information apart from the fact that the statement is true. (Wikipedia, 2021) In our case, this would mean that a voter could send their encrypted vote along with all the required

[65] People entitled to vote on national elections in 2020 (Bundesamt für Statistik, 2021)

information except the signature and the address of the voter to the smart contract via the relay, along with a zero-knowledge proof that can prove that the sender of the request indeed is a voter and hasn't voted yet. Now the contract would know whether a vote sent to it is sent from an eligible voter or not, but without ever knowing the address of the voter.

There are already some blockchain projects that use zero-knowledge proofs like z.cash[69] and Aztec[70] which allow users to make private transactions.

Using the zero-knowledge proof, voter accounts could be reused across multiple polls, which means that voters would only need to be registered once for a category of poll (e.g., national elections) and could then vote on any upcoming polls with their account. This way user accounts don't need to be regenerated and registered on the blockchain for every poll, which saves time and money (transaction fees for adding voters). Voters could also be sent an inexpensive physical security key/hardware wallet when they are first registered, which would dramatically increase security compared to using the current approach with generated letters because no one and no computer except the hardware wallet would ever know the Ethereum private key. Currently, when the access letters are generated, the Ethereum private key will first temporarily be stored on the computer that generated them, then temporarily on the device used to transport it to the printers, then temporarily on the printers and finally on the device that is used to vote during the user voting process.

The introduction of a zero-knowledge proof would also allow individual polls to be grouped much like they are on ballots in real political elections. The voters would then receive only one pair of access letters with one mnemonic, which could be used to vote on multiple polls that have been grouped. Currently, this is not possible, because it would allow voter-profiling to some extent.

The reason why I didn't implement a zero-knowledge proof in my project was that I didn't know enough about how zero-knowledge proofs mathematically work and then to run a zero-knowledge proof in the smart contract, I would have needed to write a lot of code in the Yul assembly language alongside my solidity code. Learning enough about zero-knowledge proofs, learning Yul and then implementing a zero-knowledge proof that fits my needs would simply have taken way too much time.

5.5.5 Hardware Wallets

The CLI tool and the registration app currently require the admins to enter their Ethereum private keys to do various actions crucial for the voting process, like signing generated voters. Metamask also requires the admins to import their private key to sign in to and use the admin app.

This method is for several reasons not very secure:

The admins need to enter their keys correctly for the application to work correctly, them mistyping the keys might not be obvious at the time and only be realized at a later point in time, which could lead to having to restart parts of the election process. If a registrar for example enters a private key in the registration app that is not correct, this will only be realized once they try to upload the verification hashes to the smart contract and then they need to regenerate all the voter accounts.

[69] https://z.cash/
[70] https://aztec.network/

Even though the CLI tool and the registration app only allow entering one Ethereum private key at a time, and it is assumed that only the admin that is currently entering the private key is viewing at the screen, practically there is still the chance that someone unauthorized looks at the computer's screen and gets hold of the private key this way. This can for example happen when a camera is hidden in the room where the admins meet to set up the election. If someone unauthorized gets access to the private key of a single or even multiple admins, they could use the key to execute some of the functions that are reserved for admins.

If a physical hardware wallet would be used, all the encryption and signing would be done inside the wallet, which means that the private key would never be revealed to anyone. Mistyping wouldn't be an issue anymore since the private key never needs to be entered by the admins. Someone spying on the computer's screen to get hold of the private keys also wouldn't be an issue, because the key won't ever be displayed to anyone.

Another benefit of hardware wallets is that in the unlikely scenario where someone installs a virus that would try to get access to the admin's private keys on a computer that is used to create an election, the virus wouldn't be able to get hold of the private keys, because they are never directly exposed to the computer.

5.5.6 Access Letter Encryption between Registration App and Printers

When voter accounts are generated with the registration app, they are currently stored unencrypted in the downloads folder of the computer where the registration app is installed on. This is not very secure, because anyone who gets hold of the files containing the access letters could open them and copy the mnemonics which would allow them to vote on the poll, even if they are not authorized to do so. This person could be an admin or someone who managed to intercept the file with the access letters while it was being transported to the printers.

To prevent this in a future version, the printers would generate an RSA key pair and give the public key to the registrars before they start registering users. The registrars would then enter the RSA public key in the registration app, which will then encrypt the access letters after they are generated. This way, a rogue admin or someone unauthorized who manages to intercept the access letters won't be able to get hold of the mnemonics and use them to vote on the poll. The access letters can then only be decrypted by the printers, which have the RSA private key.

The reasons why I didn't implement this feature are that I didn't have enough time and that it would have made testing even more time-consuming.

5.5.7 Rewrite Code that generates Access Letters to C++

The code that generates the access letters in the registration app is written in JavaScript and runs in an electron app. While electron makes it easy to build cross-platform apps with JavaScript, HTML, and CSS, it is not particularly fast when running resource-intensive tasks. The app has no problems generating access letters for only a few voters and does so in a few seconds. But when there are a lot of users, the app needs a lot of time. It would make sense to rewrite the resource-intensive code in C++ and then execute it in its own child process inside the electron app. C++ is faster than Node.js for several reasons, one of the main reasons being that C++ is a fully compiled language, which means that that there is no runtime parsing of source code like in JavaScript. (Baker, 2019)

5.5.8 Use ECC Keys instead of RSA Keys for Vote Encryption

ECC[72] keys are more secure than RSA keys with the same length. A standard 256-bit ECC key for example offers the same security as a 3072-bit RSA key. (PKI Consortium, 2014) Since a public ECC key is shorter than a public RSA key with the same level of security, it would be cheaper to store an ECC public key on the blockchain.

5.5.9 Use dedicated Nodes

The project currently always uses Ethereum nodes provided either by the company Alchemy or Infura to access the blockchain. This means that whenever someone, be it a voter or an admin, needs to get data from the blockchain or execute a function on the smart contract, the request is first sent to Alchemy or Infura, which will then execute the request on one of their nodes. Although it is not very likely, they could deliberately send wrong information about the contract's state and the status of transactions. This risk could be mitigated in the future if the organiser of the poll would use their own nodes, so they wouldn't need to rely on a third party. I didn't use custom nodes because I didn't have a computer with enough storage space available to store a copy of the Ethereum blockchain.

5.5.10 Better and more accessible UI

The current design of the admin and voting app isn't extremely bad and do their job. However, because I used the standard Bootstrap 4 components, it also isn't very aesthetically pleasing in my opinion. If I had more time available, I would have written a custom Bootstrap theme, which would have given the apps a more futuristic design - appropriate for e-voting apps.

5.5.11 Custom Blockchain

In a more advanced version of the system, it could also make sense to build and use a custom blockchain only for e-voting. The blockchain would still be Ethereum based but could be modified to reduce transaction costs to a minimum by using a proof of authority or proof of stake consensus algorithm instead of proof of work. There could also be a feature that would allow contracts to pay for the transaction costs of their voters, so there would be no need for a relay anymore. The transactions could be sent directly to the blockchain, which would make the e-voting system more decentralized and resilient.

A custom blockchain would also allow implementing shadow transactions in the form of a zero-knowledge proof, as described in an earlier section, directly in the blockchain's code, so smart contracts wouldn't have to implement the required code themselves.

Another feature of the blockchain could be that it would allow to generate and store RSA keys decentrally, where all nodes know the public key, but the private key would be spread across multiple validator nodes, which would only reveal their part of the private key when for instance a certain condition stated in a smart contract is fulfilled. It is not very likely that this feature can securely be implemented in a blockchain soon because there are still a lot of challenges and open questions ahead on how a system like this would be built, but for the e-voting system, it would remove the need to trust the electoral board that they won't decrypt the RSA private key before the election ends.

If a custom blockchain would be used in the future, it would need to be ensured that there will be enough nodes operated by a variety of different entities.

[72] https://en.wikipedia.org/wiki/Elliptic-curve_cryptography

5.5.12 Use different Salt than UUID

When a vote is generated, it currently uses a UUID as a salt to make the vote hash unique. This is not a good solution, because UUIDs are unique but not particularly random, which makes it easier to guess than a salt generated by a function that returns random values. That’s why in a future version a function that returns a random salt should be used instead of a UUID. It could also be considered to use multiple salting rounds.

5.5.13 Improve Component Structure inside the Admin App

Many files inside the components folder in the admin app contain multiple components. These components could be split up into individual files. The structure could also be improved by creating folders that contain the subcomponents of their respective parent components.

5.5.14 Add Tests and CI to the other Applications

Currently, only the smart contract has test cases and uses continuous integration. Adding tests to the frontends and other apps would reduce the risk of faulty software being deployed in the future.

5.5.15 Convert CLI Tool to Desktop Application

To make the functions of the CLI tool easier to access and use, it would make sense to convert it to an electron app in the future. The desktop app of the CLI tool could even be merged with the registration app, making setup and use even easier by removing a software component.

5.6 Conclusion

At the beginning of the paper, it was determined that the system should fulfil the requirements set out in the requirement catalogue of the federal chancellery along with some requirements I set out myself. The system should be able to guarantee the individual and complete verifiability of the votes, be user friendly and accessible to voters with disabilities, don't allow casting of invalid ballots and needs to keep the votes secret until the election ends. Additionally, from my own requirements, the system should also be useable for non-governmental elections and store important data and logic on the blockchain as far as possible.

Now coming back to the central question of the chapter:

Is it possible to build a decentralized e-voting system that can be used for governmental elections in Switzerland on initiatives and referendums?

Yes, it is possible to write a decentralized e-voting system that can be used for voting on initiatives and referendums in Switzerland. The system I designed would need some improvements to be actually used for real governmental elections, but the results of the requirements analysis show that the system meets most of the requirements set out by the federal chancellery and me.

The system allows voting on initiatives and referendums like they are common in Switzerland, but also voting on private elections where the voters can vote on one option out of a maximum of 5 options. The system preserves the privacy of the voters and the secrecy of the vote, while at the same time having a public and transparent voting process, which is not the case with other systems. The system allows each voter and the administrators, and in general, any interested entity, to see live how the encrypted votes are being received by the smart contract, how the votes are counted at the end of the election and much more. This means they can also check if everything is done correctly. Also, anyone can view the source code of the deployed smart contract to verify it.

The paper has shown that in the area of system design the major problems of a decentralized e-voting system are the payment of the transaction fees when casting the votes and the secure storage of the votes between casting and counting. These problems can be solved by using a relay that pays the transaction fees when the votes are cast and by encrypting the votes with a public key before sending them to the smart contract and then storing the votes in the event log of the smart contract. Furthermore, extremely high transaction fees on the main Ethereum blockchain can be avoided when using other Ethereum based blockchains, because they usually have much lower transaction fees and are also faster.

I enjoyed working on the project and did not find it easy in terms of difficulty, there were several major challenges mainly regarding system design where I really had to work for several days to find a solution. I always had to make sure that the implementation of one requirement did not compromise another requirement or the security of the system. Especially in security I always had to think in advance where the system could potentially be attacked and how this could be prevented. In the end, I would have liked to implement some of the improvement suggestions I made to make the system even better, which unfortunately was not possible because I didn't have enough time left.

I also learned a lot about Ethereum and blockchain technologies, which I find a very interesting topic where there are still a lot of opportunities for new applications and inventions. Even though I already knew react and JavaScript before starting the project, I learned a lot more about them during the project and it certainly improved the quality of the code I am writing in JavaScript. With solidity, I also learned a completely new programming language that I can also use in the future in other software projects.

6 Appendix

6.1 Bibliography

Baker, S. (2019, February 7). *Why is C++ so much faster than Javascript, but harder to code?* Retrieved from Quora: https://www.quora.com/Why-is-C-so-much-faster-than-Javascript-but-harder-to-code

Berg, P. R. (2019, August 11). *Writing Accurate Time-Dependent Truffle Tests*. Retrieved from Medium: https://medium.com/sablier/writing-accurate-time-dependent-truffle-tests-8febc827acb5

Bundesamt für Statistik. (2021, December 12). *Stimmbeteiligung*. Retrieved from Bundesamt für Statistik: https://www.bfs.admin.ch/bfs/de/home/statistiken/politik/abstimmungen/stimmbeteiligung.html

Cloudflare. (2021, November 29). *How does public key encryption work? | Public key cryptography and SSL*. Retrieved from Cloudflare: https://cloudflare.com

Edpresso Team. (2021, November 29). *What is hashing?* Retrieved from educative: https://www.educative.io/edpresso/what-is-hashing

Ethereum Foundation. (2021, November 11). *Gas and Fees*. Retrieved from Ethereum: https://ethereum.org/en/developers/docs/gas/

Ethereum Foundation. (2021, October 30). *Introduction to smart contracts*. Retrieved from Ethereum: https://ethereum.org/en/developers/docs/smart-contracts/

Ethereum Foundation. (2021, October 19). *Introduction to smart contracts*. Retrieved from Ethereum: https://ethereum.org/en/smart-contracts/

Federal Chancellery. (2014, June). Retrieved from Federal Chancellery: https://www.bk.admin.ch/bk/en/home/politische-rechte/e-voting.html

Federal Chancellery. (2021, December 12). *E-Voting*. Retrieved from Federal Chancellery FCh: https://www.bk.admin.ch/bk/de/home/politische-rechte/e-voting.html

Infante, R. (2019). *Building Ethereum Dapps*. Shelter Island: Manning Publications Co.

My Crypto. (2021, October 26). *How Do Secret Recovery Phrases Work?* Retrieved from My Crypto: https://support.mycrypto.com/general-knowledge/cryptography/how-do-mnemonic-phrases-work/

OpenGSN. (2021, November 21). *Ethereum Gas Station Network (GSN)*. Retrieved from GSN Documentation: https://docs.opengsn.org/

PKI Consortium. (2014, June 10). *Benefits of Elliptic Curve Cryptography*. Retrieved from PKI Consortium: https://pkic.org/2014/06/10/benefits-of-elliptic-curve-cryptography/

Twilio. (2021, December 9). *What is Public Key Cryptography?* Retrieved from Twilio Blog: https://www.twilio.com/blog/what-is-public-key-cryptography

Wietlisbach, O. (2019, March 12). *Das E-Voting der Post ist offiziell gehackt – so reagieren Bund, Post und die Hacker*. Retrieved from Watson:

https://www.watson.ch/digital/schweiz/775256008-das-e-voting-der-post-ist-offiziell-gehackt-so-reagieren-bund-und-post

Wikipedia. (2021, November 3). *Blockchain*. Retrieved from Wikipedia: https://en.wikipedia.org/wiki/Blockchain

Wikipedia. (2021, November 29). *Digital signature*. Retrieved from Wikipedia: https://en.wikipedia.org/wiki/Digital_signature

Wikipedia. (2021, December 12). *Zero-knowledge proof*. Retrieved from Wikipedia: https://en.wikipedia.org/wiki/Zero-knowledge_proof

6.2 Table of Figures

6.3 List of Tables

6.4 Listings

6.5 Code Repository

The source code of the project is available on Bitbucket by scanning the QR code below or entering the URL in a web browser.

URL: *https://bitbucket.org/evotesystem/evotesystem/src*

6.6 Eigenständigkeitserklärung

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig und ohne Benützung anderer als der angegebenen Quellen oder Hilfsmittel verfasst bzw. gestaltet habe.

Ort, Datum	Name, Unterschrift