AALBORG UNIVERSITY
STUDENT REPORT

**Center for Communication, Media and Information Technologies**
A.C. Meyers Vænge 15
DK - 2450 København SV
http://www.cmi.aau.dk/

**Title:**
Internet Voting: securely possible or a risk to democracy?

**Theme:**
Project report

**Project Period:**
Autumn Semester 2019

**Project Group:**
5.9

**Participant(s):**
Johannes Mols (20174921)

**Supervisor(s):**
Emmanouil Vasilomanolakis
Lene Tolstrup Sørensen

**Page Numbers:** 65

**Date of Completion:**
December 14, 2019

**Abstract:**

This project researches possible solutions to Internet Voting systems that would satisfy the security requirements of a real election. A solution is proposed that safely stores and manages votes on the Ethereum blockchain, encrypts and tallies votes using homomorphic encryption, and proves the correctness of votes and final tallies with non-interactive zero-knowledge proofs. A proof-of-concept is presented that implements Ethereum smart contracts and a front-end application to interact with the system. While we believe it is possible, we were not able to implement zero-knowledge proofs in this project. We conclude that the proposed solution is secure, but would likely fail in real-life due to the trust required by voters, which is unlikely to be given to a complex system that is hard to understand without a technical background.

# Contents

# 1 Introduction

Elections have been around for a very long time. While they were used in ancient Athens and Rome, elections have become a popular instrument beginning in the 17th century, with the emergence of representative governments in Europe and North America [1]. With most western countries being democratic today [2], whether it is direct, representative, or liberal, voting is an extremely important part of the democratic process.

Traditionally, elections are carried out using paper-based ballots. This is a system that has proven to be secure over many hundreds of years, as every conceivable method of fraud has likely been tried and detected, and since defended against. Tampering with paper-based elections is a massive effort, and many people are needed without anyone exposing the fraud.

As a result of this, virtually every democracy on earth is still using paper-based ballots. In the past few decades, technology has invaded practically every part of our society, but elections are still untouched. And that is for good reason. Electronic voting systems are much easier to manipulate, and with a single vulnerability, an entire election could potentially be flipped. With this much at stake, the security risk is simply not worth it.

However, some countries use electronic voting despite the risk. Especially with Internet voting instead of using electronic voting machines, there are many advantages: it is cheaper, faster and more efficient at counting votes, and could bring higher voter turnout [3]. For example, Estonia became the first nation in the world to hold legally binding general elections over the Internet in 2005 [4]. Even though no attacks on the Estonian system have been detected as of today, many security experts have condemned the system, including a group of researchers that were able to manipulate votes and vote totals by installing malware on both personal computers and the voting servers themselves [5].

This project examines the advantages and disadvantages of electronic elections via the Internet and voting machines from a security perspective, and whether it is possible to build an electronic system that is equally or more secure than traditional paper-based elections. A prototype solution for Internet voting is presented in this project, which takes into consideration the vast amount of research about the topic, and builds on top of it. After presenting the prototype, it is then discussed whether it could potentially be used for secure elections, or if it is not a good idea to use electronic voting after all.

## 1.1 Research question

Is it possible to design an Internet voting system that can match the security of paper ballots? What are the requirements for such a system and how can they be fulfilled in a scalable and practical way?

# 2 State of the Art

Elections are worthless without the consensus among voters that their votes are cast securely and anonymously, and that the final count accurately represents the will of the voters. Three main technologies are widely used today: paper ballots, electronic voting machines, and voting over the Internet. The following section provides an overview over of each of them, describing advantages and disadvantages.

## 2.1 Paper Ballots

Paper-based voting systems are the most secure way of conducting an election. This is due to two important aspects [6]:

- Voters know that the ballot accurately represents their intent because they can examine the ballot themselves
- Voters know that there is a physical record of their vote, which is difficult to destroy or change without leaving physical evidence

This is what sets paper-based voting apart from digital systems, and that is why the majority of the world still completely relies on paper ballots in their elections.

Despite this, there are significant disadvantages to paper systems. One of them is cost. Taking Germany as an example, the German government has spent 92 million Euros on the general elections in 2017 [7]. The main factors for this are the cost to send election notifications to eligible voters, and actual ballots used to vote via mail. The second cost factor is the compensation of election helpers. Germany employed 650 thousand helpers in the 2017 election, and each of them received 35 Euros for a full day of work. Accounting for just one day, that is 22.75 million Euros.

The second disadvantage was already mentioned: the amount of helpers required to pull it off. Germany employed 650.000 citizens in the 2017 election, which is almost 1% of the population. While election helpers are probably happy to do it for the sake of democracy, it isn't ideal. Finding a way to reduce the number of helpers needed would be a great relief.

Lastly, it it is inefficient and takes time. Sending ballots per mail, and manually counting every single ballot is time-consuming. Audits are only possible by manually re-counting a statistical significant sample by hand.

All of this is worth the price, and every democratic government spends the required money for a fair election. However, not every country has the capability to employ enough election helpers, or don't want to spend the money. Because of this, countries with high populations such as the United States, Brazil and India make heavy use of Electronic Voting Machines, which are discussed below.

## 2.2 Electronic Voting (e-Voting)

Electronic Voting Machines are mechanical or electronic machines that take care of casting and counting votes. They have a long history in especially the United States, where the first voting machines were invented. The first voting machines were introduced around 1890, with the lever voting machine, and the punched card system. Lever voting machines keep the total of votes by a mechanical wheel, making recounts impossible. Punch cards are pieces of papers that can be punched with a hole relating to a candidate. Those can then be counted with a punched card reader, tabulating machine, or optical scanners. Both of those systems have proven to be insecure and unreliable, and are not used anymore. However, modern voting machines suffer from equally serious security issues, as the following overview will show.

The most commonly used EVM's in the United States are Direct-Recording Electronic Voting Machines (DRE). While other countries such as India use different designs for their EVM's, it is safe to say that both designs are insecure, as an independent security analysis of Indias EVM's showed [8]. In the following, we will mainly focus on the general risks of those machines, and not specific examples, as there are too many different machines and designs.

There are many requirements for a fair election, and there is no consensus among researchers which requirements are necessary. But a paper on the risks of e-Voting gives a good list of the main aspects of securing an election [9].

- The voter must be authorised and authenticated
- Eligible voters are citizens above a certain age
- Voter must be able to vote anonymously to prevent vote selling and voting under coercion
- Votes must be assured confidentiality (secret) and integrity (recorded as intended)
- Voting technology must produce logs to allow audits
- Voting machine configuration must be open and controlled to allow inspection
- Voting machine can't be modified while operating
- Voting machine must be reliable and available
- Persons responsible for developing, operating, and maintaining machines should have records of impeccable behaviour

The paper identifies a number of vulnerabilities that DRE voting machines are exposed to. These can be seen in Figure 1. At every stage, from developing the machines, to actually using them, there are several severe security vulnerabilities that should discourage anyone of using DRE's.

The American Brennan Center for Justice released a fact sheet about voting system security, stating that in 2016, 42 out of the 50 US states were using voting machines that are at least 10

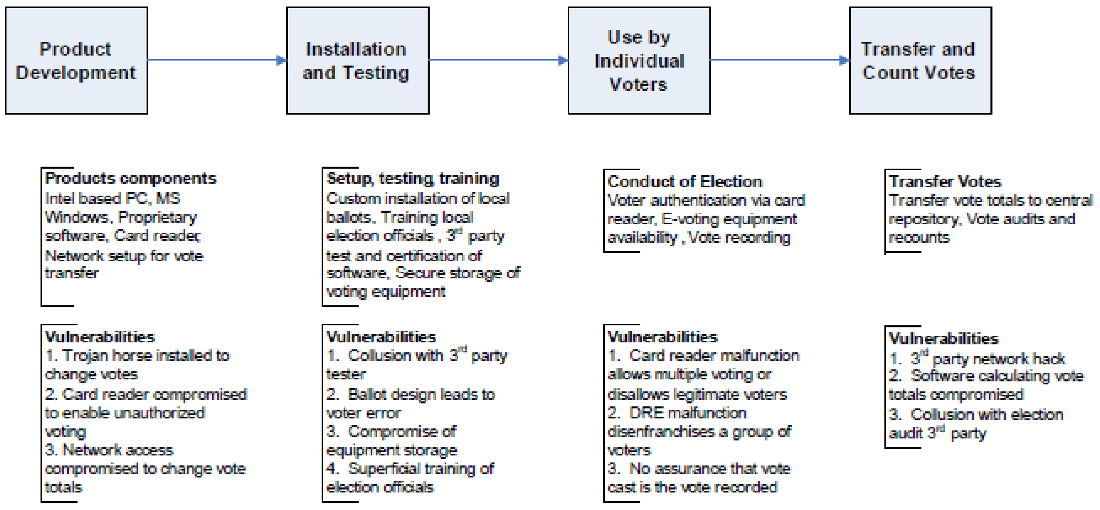

Product Development
Installation and Testing
Use by Individual Voters
Transfer and Count Votes
Products components
Intel based PC, MS Windows, Proprietary software, Card reader, Network setup for vote transfer
Setup, testing, training
Custom installation of local ballots, Training local election officials, 3rd party test and certification of software, Secure storage of voting equipment
Conduct of Election
Voter authentication via card reader, E-voting equipment availability, Vote recording
Transfer Votes
Transfer vote totals to central repository, Vote audits and recounts
Vulnerabilities
1. Trojan horse installed to change votes
2. Card reader compromised to enable unauthorized voting
3. Network access compromised to change vote totals
Vulnerabilities
1. Collusion with 3rd party tester
2. Ballot design leads to voter error
3. Compromise of equipment storage
4. Superficial training of election officials
Vulnerabilities
1. Card reader malfunction allows multiple voting or disallows legitimate voters
2. DRE malfunction disenfranchises a group of voters
3. No assurance that vote cast is the vote recorded
Vulnerabilities
1. 3rd party network hack
2. Software calculating vote totals compromised
3. Collusion with election audit 3rd party

Figure 1: DRE life cycle showing security vulnerabilities by stage [9]

years old [10]. Those machines use outdated hardware and software that don't receive regular security patches (e.g. Windows XP and Windows 2000). Many of those machines also don't have a federal certification, and 14 states use machines that do not produce a paper record, making it impossible to audit. Furthermore, voting machines are notoriously easy to hack. This is demonstrated every year at the DEFCON Voting Village, which is an annual hacker convention in Las Vegas. For the third year in 2019, researchers at the DEFCON Voting Village were able to compromise over 100 commercially available voting machines used in current US elections [11].

Tom Scott, a British entertainer and educator, released a popular video in collaboration with Computerphile, a YouTube channel dedicated to topics of Computer Science, titled *"Why Electronic Voting is a BAD Idea"* [12]. Scott argues that Electronic Voting Machines can never be trusted, and should never be used, because they are essentially a black box that voters submit their vote with, and spit out a number at the end of the process. Auditing the software and hardware of voting machines is not possible, because they are often developed by private companies and use proprietary source code. Even if the source code was openly accessible, Scott argues that one still can't trust that the machines are actually running the correct software. Furthermore, voters have to be left alone in a booth with the machine in order to assure an anonymous vote. However, the previously mentioned DEFCON conference has shown that one minute with the machine is enough to attack it and potentially install malicious software via exposed hardware. Lastly, Scott argues that tally results are often transmitted to election centers over insecure connections, which could easily be manipulated. All of these concerns may also apply to i-Voting.

To conclude on the topic of Electronic Voting Machines, this review has made it clear that EVM's are full of vulnerabilities and are easy to compromise. The mentioned research is only the tip of the iceberg of an enormous amount of information about the security risks of EVM's. Because of this, we will not consider Electronic Voting Machines to be a potential solution for this project, and any election system in general.

## 2.3 Internet Voting (i-Voting)

Internet Voting is another type of voting that has risen in popularity in the past years. While there are only few cases of real elections held over the Internet, many polling services exist on the web (e.g. Strawpoll, Doodle, Google Forms). In the following, we will review different types of i-Voting systems that are designed to be used for actual elections, as those have a much higher need for security.

### 2.3.1 Estonian Internet Voting System

Estonia was the first country in the world to hold legally binding general elections over the Internet, and is still using the system today. Estonia is one of the most advanced digital society in the world, with 99% of government services being available online [4]. And so is their elections, with approximately 44% of voters voting from home. The following section provides an overview of the system and the security concerns.

#### 2.3.1.1 Legislation and Principles

The Estonian election legislation has the following requirements for i-Voting [13]:

1. On advance polling days, voters holding a certificate for giving a digital signature may vote electronically on the web page of the National Electoral Committee. A voter shall vote himself or herself.
2. A voter shall identify himself or herself by giving a digital signature.
3. After identification of the voter, the consolidated list of candidates in the electoral district of the residence of the voter shall be displayed to the voter on the web page. The opportunity for the voter to examine the national lists of candidates shall be provided.
4. The voter shall indicate on the web page the candidate in the electoral district of his or her residence for whom he or she wishes to vote and shall confirm the vote.
5. A notice that the vote has been taken into account shall be displayed to the voter on the web page.

Additionally, the online elections are held 4 to 6 days before Election Day, which gives voters the chance to vote again on Election day and invalidate their i-Vote. An important principle is that a traditional vote always has the priority over an online vote. Having the online election prior to Election day also makes it possible to invalidate all online votes in case there are reasons to believe the system has been compromised. Traditional elections are still held to avoid a digital divide in the country. With more voters using i-Voting each year however, it could possibly replace traditional elections entirely in the future.

Another important principle is that voters are allowed to change their vote during the online election period. This is done to prevent vote selling and voting under coercion, which is much more likely when voters can submit their vote from home instead of a secure voting booth.

#### 2.3.1.2 Implementation

The goal of the system was to design it in a way that isn't too complex to understand for regular voters. It was feared that if the system is too complex, voters wouldn't trust it because they can't understand how it works. Because of this, the system uses an *Envelope method* which can be easily compared to a regular ballot envelope used for absentee voting.

In a traditional vote via mail, the voter first identifies themselves to an election authority, which sends the voter a ballot. The voter fills out the ballot and puts it in an inner envelope. This inner envelope is then put into another envelope that contains identifying data such as name and address. When the envelope arrives at the polling place, authorities verify the voter's identity from the outer envelope, and the anonymous inner envelope is put into the ballot box.

The i-Voting system uses a very similar approach. When voting from home, voters identify themselves by inserting their ID-card with electronic features into a card reader. A list of relevant candidates is displayed on the Electoral Committee's website. The voter's choice is encrypted with a public key that everyone uses. This is the inner envelope. A digital signature is created with the help of the ID-card. Both the encrypted vote and the digital signature is sent to the Electoral Committee's servers. On Election Day, the digital signature is removed and the encrypted votes are collectively decrypted using the private key that the Committee keeps secret. When the voter now votes again on Election Day, the polling stations have a list of all identities that participated in the i-Voting and can request the Committee to remove the i-Vote because the voter has submitted a new vote using paper ballots. A diagram of this method can be seen in the Figure 2 below.

#### 2.3.1.3 Concerns

A group of international i-Voting experts have observed Estonia's local elections in October 2013 and released a detailed report with their findings [5], as well as material such as videos and photos on a website (`https://estoniaevoting.org`). This subsection summarizes their findings.

The group *"observed operations, conducted interviews with the system developers and election officials, assessed the software through source code inspection and reverse engineering, and performed tests on a reproduction of the complete system in their laboratory"*.

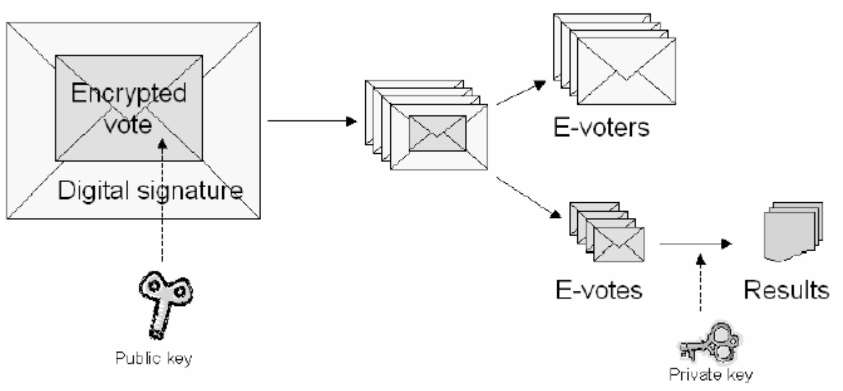


Figure 2: The envelope method used in Estonia's i-Voting system [13]

Their findings are concerning, and manipulating the election is certainly possible for a sophisticated attacker. The researchers have found serious weaknesses both in the architecture of the system, and the procedures.

In terms of architecture, the Estonian system is not end-to-end verifiable, which means that that voters' computers, servers, and election staff have to be implicitly trusted for the system to work. E2E verifiability would allow anyone to confirm that ballots have been counted accurately without having to trust the computers or election officials. To achieve the required trust, Estonia uses a set of operational procedures.

However, while observing the election in 2013, the researchers found that the procedural controls were inadequate and not consistently followed. They observed that procedures seemingly changed every day, staff restarted server processes and ignored error messages at their own discretion, and asking staff members about procedures sometimes yielded indecisive answers.

Furthermore, operational security was less than sufficient. The observers noticed that election staff downloaded software over unsecured HTTP connections from public websites, and then used personal computers to set up election servers. This could allow attackers to insert malware in the downloaded software or the personal computers itself. Furthermore, election workers openly showed root passwords and ID-card PIN codes on camera. Physical keys to the server rooms were also shown on camera, potentially allowing criminals to duplicate them. The server rooms had no 24/7 security personnel, it was unclear who monitored the cameras, and workers created unencrypted daily backups on DVDs that were transported in personal backpacks. Personal USB sticks were used to transfer official votes to the counting server, which could easily contain malware. The amount of possibilities to abuse these security issues are alarming.

While election officials allow public observation, release videos of operator tasks, and release the majority of server-side source code, the researchers don't think that this is sufficient. First, allowing public observation and releasing videos does not prove that malicious operations couldn't be performed before the observation started. They also argue that the entire source-code needs to be released for it to be trusted. The reason that source code for the client application was not published is that officials are concerned that it would make it much easier to create fake software. However, the researchers have shown that this is easy enough without the source code.

On top of that, the group of researchers created a mock election setup with the published source code, and demonstrated how they could compromise both the client-side and server-side of the election.

On the client side, the researchers developed a custom piece of malware that runs on a voter's personal computer and records the PIN code of the ID card when the user is submitting their original vote. The malware then runs silently in the background and replaces the vote at a later point, even bypassing the mobile verification app if the voter uses it.

On the server side, the researchers were able to inject malware in a variety of ways onto the counting server, which could modify the results freely. Another vulnerability that the researchers are concerned about is zero-day exploits such as the OpenSSL Heartbleed bug, which the observed servers were vulnerable to.

The researchers conclude that Estonia should withdraw the online voting system, as there are too many vulnerabilities in the basic architecture and procedures to fix [5].

#### 2.3.1.4 Conclusion

This review of Estonia's i-Voting system shows that it is possible to develop a relatively secure system, but it is extremely difficult, if not impossible, to defend against every possible attack. Estonia has shown which legislation is needed and what requirements the implementation should have at the very least, which is helpful to identify requirements for this project.

The biggest take-away from this review is that election officials, personal computers, and voting servers cannot be trusted. Any secure i-Voting system needs to implement end-to-end verifiability and should be designed so that election staff only have to perform as little tasks as possible on the system, which all need to be fully transparent. In Estonia, election workers have access to all the crucial infrastructure and could perform an insider-attack without much resistance.

It is also crucial to be aware of malware that might run on the unsecured personal computers of voters, or potentially be installed on the election servers. It has been shown that this could compromise single votes when installed on voter's computers, and manipulate an entire election if installed on the servers.

**Note:** All the following projects are based on Blockchain technology, all of which are using Ethereum. See section 2.4.1 for an introduction to Blockchain and the Ethereum project.

### 2.3.2 BroncoVote

BroncoVote is a university project made by students in the USA. It implements a secure voting system over the Internet by using Ethereum's blockchain and smart contracts [14]. Besides the security of the system, the project also focuses on privacy by encrypting the votes.

The system uses three separate *smart contracts* (see introduction to Ethereum, section 2.4.1) that are deployed on an Ethereum blockchain. This is the back-end of the system, where the logic of the voting process is located. The nature of the blockchain provides the needed security that can't be provided by regular servers, as we saw with Estonia's i-Voting system (see section 2.3.1).

The three smart contracts have the following responsibilities:

- *Registrar*: keeps a list of all registered voters and ballot creators
- *Creator*: creates *Voting* contracts for every new election, with details such as time limits and a list of options
- *Voting*: acts as a virtual ballot and allows voters to cast their vote

For the front-end side of things, BroncoVote provides a simple single-page web app built with HTML and JavaScript. Here, users can create elections and vote. There are three different types of users who can interact with the system:

- The *administrator* deploys the initial smart contracts and can grant ballot creation permission to other users
- The *creator* is a voter who can create elections
- The *voter* can vote in an election given that he is registered

To secure voter's privacy, BroncoVote makes use of a cryptographic concept called *homomorphic encryption.* This is an asymmetric encryption scheme that allows computation on ciphertexts. This means that voters can encrypt their votes with a given public key, and submit that to the smart contract. The tally can then be calculated by multiplying every ciphertext (multiplying ciphertexts is the equivalent to adding plaintexts). The final, encrypted number of votes can then be decrypted without revealing how anyone voted.

#### 2.3.2.1 Issues

While the project has really good ideas, there are also some concerns about it.

First of all, it utilizes a server that handles encryption and decryption of votes. This is because Ethereum contracts can only handle up to 256-bit integers, and asymmetric encryption keys should be at least 2048 bits long [15]. To circumvent this limitation, they use an unprotected server, which they acknowledge to be a security vulnerability. A perfect example of this problem has actually just been shown in Moscow's City Duma election in September 2019. An Ethereum-based system was used to cast votes, and the votes were encrypted using an asymmetric encryption scheme with 256-bit long keys. A french researcher released a report shortly before the election that shows how the private key can be computed from the public keys within a matter of minutes. It is unclear why 256-bit keys were used, but the researcher assumes it is due to the limitation of the Ethereum smart contracts, just as it is in this project [16].

Furthermore, the project was created for a university setting where there wouldn't be a high number of participants and security isn't as big of a concern as for national elections. That is why the registration process is not secure. In order to register as a voter, you have to provide an e-mail address with a specific domain (the university's e-mail domain in that case). There are no verification e-mails sent to verify the correctness of the address. Obviously, that isn't good enough if this system should be used for larger elections with higher security requirements.

The authentication of voters is also highly flawed. The BroncoVote paper shows that the client-side application performs the authentication by sending a request to the *Registrar* contract. If a positive answer is received, the application proceeds to send the vote to the voting or creator contract. This can easily be abused by creating a custom application that skips the authentication and sends malicious requests to the contracts right away. A look at the source code confirms this, there is no protection on the relevant functions. Authentication needs to happen on-chain.

And lastly, there is no verification of the correctness of votes. If the encryption server is compromised or uses modified source code, it could change votes without anyone noticing. The voter sends his unencrypted vote to the server, which then outputs an encrypted version of that vote. There is no way for the voter to verify that the output is the correct encryption of his vote.

### 2.3.3 Hääl

Hääl [17] is another Ethereum-based voting protocol that has similar ideas like BroncoVote, but is more sophisticated. It also uses homomorphic encryption to secure voter privacy, and zero-knowledge proofs to verify the correctness of encryptions and decryptions without revealing any information. This is an ideal solution to the issue of not being to able to verify the correctness of votes and the election result that we mentioned when discussing concerns with BroncoVote (see section 2.3.2.1).

Zero-knowledge proofs are a method to prove the possession of some knowledge to another party, without sharing any information except that the prover has the specified knowledge. Simply revealing the the information to the other party does work, but is not desired in many cases such as the voting application here. How zero-knowledge proofs work in detail is highly mathematical and will not be explained here.

On top of that, stealth addresses are used to allow voters to cast their ballot without anyone being able to see the transactions. Normally, every transaction on a blockchain is public and given a specific address, anyone can see all of the transactions from and to that address. With stealth addresses, an address is published which can be used derive a new address. It is impossible to know that those addresses are associated, so that transactions can't be followed. Hääl uses this to allow voters to submit their ballot without anyone knowing that they did. While the votes are encrypted, this gives another layer of privacy in that no one can know if a specific person has voted or not. With BroncoVote and other projects, everyone can know who voted, but don't know what they voted due to the encryption.

#### 2.3.3.1 Issues

This project provides more security than BroncoVote with the use of zero-knowledge proofs and stealth addresses. However, there are still issues to be concerned about.

First of, from briefly looking at the smart contract code, we noticed that it might be possible to add addresses to the list of approved voters without permission, as there is no protection on the relevant function. We contacted the creator about it but haven't received an answer yet [18].

Furthermore, voter authentication doesn't seem to be a topic for this project either, as it isn't very sophisticated. For a real-world voting system, there would need to be a secure way of authenticating voters.

There is also no front-end application provided to interact with the blockchain, as it is a proof-of-concept protocol. It also doesn't specify whether it would be deployed on public Ethereum blockchains or other, private networks.

### 2.3.4 Polys

The Russian cybersecurity company Kaspersky launched a start-up called *Polys* that is building a secure online voting system with the help of blockchain. It is the only project we could find that has actually implemented a solution in a fully working state. Anyone can create a demo vote on their website `https://polys.me`.

The company released a whitepaper [19] detailing the algorithms they use to secure the elections. It explains that Polys is using *Shamir's Secret Sharing* scheme to distribute keys, as asymmetric encryption is used to encrypt the voting results. To actually encrypt the votes, Polys is using *ElGamal's* encryption scheme, which has homomorphic properties similar to the *Paillier* scheme used in *BroncoVote* and *Hääl*. Furthermore, the whitepaper states that Ethereum smart contracts are used as the backbone of the system, just like the two projects reviewed previously.

#### 2.3.4.1 Issues

The service works very well. We created several mock elections, and it just works. The interface is intuitive and pretty, which makes it easy to use for actual voters.

However, it can not be trusted in our opinion. In the FAQ on Polys' website, the company states that they are preparing to publish the source code on a GitHub repository in the near future, and that they are currently auditing the code internally [20].

This was more than two years ago. On the GitHub repository where it is supposed to be published [21], they state that they are "resolving some legal issues" and hope to publish the source code in the near future. This update was 11 months ago as of writing this.

Without wanting to allege malicious intent to the company, it is impossible to trust the service if no third-party can audit the source code. And even if, it might not be possible to verify that the correct code is running during an election because everything is conducted through their closed-source website and servers.

Besides that, the company uses a private Ethereum blockchain, where representatives serve as miners [22]. This might open up possibilities to attack the network if the network consists of only a small number of nodes. For example, if more than 50% of miners are malicious, they could disrupt the network and prevent any new transactions, as well as reverse previous transactions. This is a common security concern with blockchains and is called a *51% attack* [23]. There has not been a recorded case of this happening yet in large networks such as Bitcoin and Ethereum, but it is possible, and a genuine concern with private blockchains where miners are representatives running for the election. Blockchains, especially cryptocurrencies, with less popularity regularly suffer from 51% attacks.

Furthermore, the U.S. government has banned the usage of all Kaspersky products in all federal agencies amid concerns that the company was working on secret projects with the Russian government. The U.S. alleged that Russian hackers stole confidential data from NSA computers via Kaspersky antivirus software. Whether this is true is unknown. Because of this, Kaspersky is not going to try and enter the U.S. market with Polys [24].

## 2.4 Relevant technologies

### 2.4.1 Blockchains and Ethereum

The blockchain technology was conceptualized by an unknown person under the alias of Satoshi Nakamoto. His or her famous whitepaper *Bitcoin: A Peer-to-Peer Electronic Cash System* [25] was published in 2008, and the proposed network called *Bitcoin* launched a year later.

Today, the Bitcoin system which is used for financial transactions is highly popular and its market cap is well over 100 billion US Dollars, with the value of a single Bitcoin being worth over 17.000 USD in December 2017.

Since Bitcoin's launch ten years ago, many hundreds of alternative cryptocurrencies entered the market, some successful, but most of them not gaining much of a following.

#### 2.4.1.1 Blockchain

Blockchains are essentially distributed databases. The same copy of the database is shared across a peer-to-peer computer network, which makes it very difficult to change existing records in the database.

As the name suggests, the database is constructed by chaining blocks together. A block is a collection of records or transactions. A transaction in Bitcoin is a transfer of a certain amount of Bitcoin from one account to another. Every transaction is checked by the network to ensure that it is valid and then added to the next block in the chain. Each block has a unique hash calculated from all the transactions in the block. It also contains the hash of the previous block in the chain. This property makes it so difficult to modify the blockchain because when an attacker tries to change a detail of a transaction, its block gets a new hash. The attacker then has to calculate the new hash for every block after that so the chain isn't broken, and then convince everyone else in the network that this is the correct copy. This is almost impossible in large networks such as Bitcoin and Ethereum.

Since there is no central authority to decide on the correct copy of the blockchain and to accept new transactions, there has to be a consensus mechanism. This mechanism has to solve the so-called *Byzantine generals problem.* The name comes from an ancient example, where two byzantine generals want to attack a city, but can only succeed if both armies attack at the same time. They are on opposite sides of the city and need to send a messenger through the city with the information on when to attack. But the problem is that the messenger could get caught and the message modified, before using their own messenger to deliver the fake message.

To solve this problem, Bitcoin uses a concept called *Proof of Work*. This essentially requires the sender to solve a mathematical problem before being allowed to send the message. For this, a *nonce*, which is a random number, is appended to the message. The hash of the message is required to start with a certain number of zero's. The sender then increases the nonce until he finds a number which results in a hash with the required number of leading zeroes. If the message is then intercepted by an attacker, and the attacker changes the message, the hash also changes. The attacker then needs to find a new nonce value that results in a hash with the required leading zeroes. This makes it very difficult for an attacker to modify a message, especially with transactions bundled together in blocks.

The verifying of transactions, also called *mining*, requires an immense amount of computational power. It has become such a big issue, that Bitcoin has a measurable carbon footprint. In 2019, Bitcoin has an annual electricity consumption of up to 45.8 Terawatthours, corresponding to carbon emissions of up to 22.9 Megatons of $CO_2$ annually [26]. Bitcoin consumes more power in a year than entire countries.

Because of this, a number of different consensus mechanisms have been envisioned (e.g. Proof-of-Stake, Proof-of-Authority, etc.). Ethereum, the second biggest cryptocurrency behind Bitcoin, wants to implement Proof-of-Stake in an upcoming release, and several Ethereum test networks already use Proof-of-Authority for consensus.

#### 2.4.1.2 Ethereum

Ethereum is a blockchain-based computing platform similar to Bitcoin. It has a cryptocurrency called *Ether* which is used to compensate miners for performing computations. Ether can be transferred between accounts just like with Bitcoin and any other cryptocurrency.

The difference that makes Ethereum so special is the ability to execute scripts on the network. This is nothing new, as Bitcoin uses *Bitcoin Script* to perform transactions. However, Ethereum uses an instruction set that is Turing-complete, allowing anyone to create *smart contracts* that can perform calculations and store data on the blockchain in a controlled manner.

Ethereum smart contracts are programmed in Solidity, a object-oriented high-level programming language, which can then be deployed as an account on the blockchain. The smart contract defines the behaviour of the account. Users can send transactions to a smart contract containing data and possibly also an Ether value. Taking the voting subject as an example, it is possible for users to send their vote as a transaction, which the smart contract can then work with. In a very basic manner, it could maintain an integer counter that increments with each vote. Since this is on the blockchain, this value can only be manipulated by the smart contract itself and a complete history of values is recorded on the chain, making it transparent and verifiable.

#### 2.4.1.3 Possible attacks

Even though blockchains are cryptographically highly protected, they are not immune to attacks. The following list outlines some of the most common attacks that are known today, and how dangerous they can be [27, p. 156].

- Blockchain
  - *51% Attack.* A mining pool with the majority of computational power in a network could control approval of new transactions, reverse previous transactions, and double-spend money. This attack only works in Proof-of-Work systems. Multiple smaller cryptocurrencies have been attacked this way, but larger networks such as Ethereum and Bitcoin are likely too large to be a feasible target.
  - *Denial-of-Service Attack.* Nodes are bombarded with transactions and slow down the processing of regular transactions. To prevent this, Bitcoin and other blockchains introduced transaction fees which make DoS attacks very expensive. Furthermore, transaction rates and maximum block sizes regulate the traffic. The time needed to mine a block in Bitcoin is 10 minutes, while it is 15 seconds in Ethereum. The Proof-of-Work difficulty is constantly adjusted to maintain the block time.
  - *Sybil Attack.* Attackers create many full nodes to isolate parts of a network, so that some legitimate nodes are only connected to the attackers nodes. By feeding the legitimate node fake transactions, double-spending is possible. This also only applies to smaller networks, as examples like Bitcoin and Ethereum are too large to be attacked. To prevent this attack, a reputation system could require nodes to build a reputation before allowing them to participate in the network. Reputation systems are used in Proof-of-Stake mechanisms.
  - *Finney Attack.* The attacker must be a miner. He can mine a block that includes a transaction from his account to another account owned by himself. This block is not broadcasted to the network. The attacker now sends another transaction to his victim, which might be a seller of something. When the seller sends the product to the attacker, the attacker broadcasts the previously mined block to the network, which invalidates the transaction to the seller because double-spending is not allowed. A case of this attack happening has never been recorded, but is theoretically possible. The timing must be very good and the attack can be avoided if the seller waits longer until the transaction is fully confirmed by the network.
  - *Selfish Miner Attack.* A mining pool could mine a block but withhold it from the network. More blocks are mined and added to the chain without notifying the network. After a while, the list of mined blocks is broadcasted to the network and will be accepted because it is the longest chain available, invalidating previous versions of the blockchain. This could earn miners more money with transaction fees. This becomes dangerous if a mining pool has a third of the network's computational power.

- Smart contracts
  - *Source Code Vulnerabilities.* Smart contracts can have vulnerabilities in their source code if not tested carefully or bugs in the underlying structure occur. Once a smart contract is deployed to the blockchain, they can not be changed. The most popular example of this is the *Parity* bug, where a user accidentally triggered a bug in the contract's code, which rendered the functions within that contract unusable [28]. Subsequently, 170 million US Dollars from 573 owners were frozen in that smart contract's account and no one can access it to this day.
- Users
  - *Phishing.* Fake websites can be used to make people believe they are using a popular website related to cryptocurrencies. They often look identical to the real websites and ask users for their private wallet keys by pretending to have a legitimate purpose for it. In reality, the private keys are immediately transferred to the attacker which can then empty the victim's wallet.
  - *Dictionary Attacks.* Very similarly, attackers can generate lists of private keys by hashing common passwords. The public address can be derived from the private key, and automated scripts can check the public blockchain if that account has any amount of money on the account. If so, it can simply be stolen because the private key is known. While it is very unlikely to guess a private key by randomly guessing characters, hashing common passwords is a method that earned attackers a lot of money over the years. Users that might not know much about the technology could have created their account with their regular password and thought that was safe. To securely create an account, a 12-word mnemonic is usually used to generate a private key. The 12 words are easy to remember, but very difficult to randomly guess. They serve as a backup seed phrase in case the private key got lost.

Many of the mentioned attacks are only theoretical and have not been knowingly used yet, but the threat is always present. Many millions of dollars have been stolen from cryptocurrencies using some of these attacks, and there are always vulnerabilities that could be exploited. Anyone developing blockchain applications should keep those attack vectors in mind.

### 2.4.2 InterPlanetary File System (IPFS)

The *InterPlanetary File System* is a peer-to-peer hypermedia protocol that is aiming to replace HTTP and provide an open and decentralized web. The argument of a distributed web in favour of a centralized one is that the modern web relies heavily on central servers, which can be inefficient, expensive and allows governments and companies to censor content easily. Regular client-server infrastructures can be congested if too many people are trying to access content from a server, which is also the phenomenon Distributed-Denial-of-Service (DDoS) attacks take advantage of.

IPFS is trying to solve many of the modern web's problems with this protocol. In the IPFS network, every file is associated with a cryptographic hash, making it impossible to alter data without changing the hash. This makes the protocol *content-based* instead of *location-based.* When requesting a file from a server over HTTP, the domain that is displayed in the browser resolves to the location of the server. In IPFS, files are requested by providing the file's hash instead of a server which can serve the file. The network is indexed by *Distributed Hash Tables*, which keep information on which nodes have which files. This means that content can be delivered much faster through IPFS if there is a geographically nearby node that can send the user the requested file. In centralized infrastructures, requests often have to travel thousands of kilometres before reaching the desired server.

The nature of the network makes it impossible to censor content. IPFS has been successfully used against censorship in the past. For example, when Turkey banned Wikipedia in 2017, users started hosting a version of Wikipedia on IPFS [29].

Some drawbacks of the network is that there is currently no such thing as private content. Anyone in the possession of the hash of a file or directory can also access it. It is possible to encrypt data to prevent unwanted access, but in its nature, IPFS is public. Another drawback is the availability of files. The network does not guarantee availability, and nodes only store files that they are interested in. If the last node hosting a specific file goes offline, it is not accessible anymore. To avoid this, files can be "pinned" on a node so that it always keeps it available. There are also many pinning services available that keep files available for money. The creators behind IPFS are also working on a cryptocurrency which would reward users for running nodes on their computers and renting out space on their hard drive for random content. The more people that engage in such an economic system, the more bandwidth and availability is available to users, and the need for content delivery networks would be drastically reduced in cases where IPFS is used.

As for security, IPFS protects against Sybil attacks (creating many malicious nodes to cut off parts of the network) by requiring new nodes to solve Proof-of-work puzzles before being allowed to join [29]. This makes the creation of many nodes expensive and infeasible in a large network. The network also naturally protects against DDoS attacks because a user has many options to retrieve a file. Attacking every single node that hosts a file is infeasible, especially with popular files.

## 2.5 Summary

This State-of-the-Art section was helpful to get an overview of current solutions and how they designed a supposedly secure system. It became apparent that paper ballots are still by far the most-used method for elections and are also the most secure since it is so difficult to manipulate paper ballots on a large scale.

Electronic Voting Machines are widely used in many countries such as the U.S., India and Brazil. However, this doesn't mean that this is a secure method of conducting elections. The opposite is true. It has been shown time and time again how easy all of the most commonly used EVMs are to compromise.

So, the only choice for an electronic voting system that can match the security of paper ballots has to be over the Internet. And there are hundreds of approaches to this problem. Reviewing the only i-Voting system used for national elections in the world, Estonia has shown that it is possible to build such a system, but not without flaws. International researchers have found many vulnerabilities, mostly consisting of human error and malware, that could potentially compromise the entire election. The researchers recommend Estonia to stop using the system immediately.

In the past decade, blockchains have gained massive popularity, mostly by the introduction of Bitcoin and it's explosion in value in 2017. Many think that it is the most impactful invention since the Internet, while others think that the hype is not justified. Either way, it has properties that make it an excellent choice for many security applications. The immutability of the chain makes it ideal to store votes.

We reviewed several projects of various scopes in this section, all of which are using the Ethereum project and its smart contract abilities (see section 2.4.1). All of the reviewed projects agreed on a similar architecture and usage of cryptographic schemes. Homomorphic encryption was used in all cases to provide voter privacy, and one project used zero-knowledge proofs to verify the correctness of votes and end results.

In this project, we will also use Ethereum's smart contracts to build the core logic of the system. While the idea for this project isn't new, as seen from the vast amount of research available, this project is. All of the code used in this project is created from scratch, and not inspired by the reviewed projects are any other source. What we are getting from these projects reviewed here is the approach they took to designing their solution, and what issues they faced.

Having said that, this project will not only focus on building a secure Ethereum back-end like most projects reviewed here, but also on building a front-end that makes it easy to interact with the system. This front-end should provide a good user authentication system, as well as countermeasures against attacks from foreign actors (e.g. Denial-of-service attacks).

# 3 Hypothesis and Objective

The objective of this project is to build a secure Internet Voting System. In the previous section, we have reviewed many different types of voting systems that exist today, and learned what works and what doesn't.

Like many of the reviewed systems, this project also uses the Ethereum project to build a blockchain voting system. The State-of-the-Art has shown that this is a good approach that the literature supports. This project makes use of three Ethereum smart contracts, each with a specific purpose. One contract is controlled by a special registration authority that maintains a list of eligible voters in the smart contract. The second smart contract is controlled by an election authority, and it is responsible for creating new instances of the third contract, the actual Election contract, and to keep a list of all elections for reference. The Election contract is where the actual voting happens. A start and end time, as well as a list of candidates is provided before the election starts. During the election, nothing about the details can be changed anymore. Voters can now submit their vote to the contract. After the time runs out, no new votes are accepted and tallying can begin.

The usage of three smart contracts is similar to the structure of *BroncoVote* (see section 2.3.2). The high-level architecture is similar, but the inner workings of both systems are quite different. Many flaws described in the review are solved by this project's approach (e.g. usage of external encryption server, easily bypassed authentication).

As for the security aspect, the entire system will be decentralized to avoid the security risks of centralized servers. The Ethereum blockchain itself is decentralized and hosted by thousands of nodes around the world, which makes it tamper proof and secures the sensitive voting data. The second part of the system, the web-based front-end application, will also be decentralized using the *InterPlanetary File System* (IPFS, see section 2.4.2). This will make it much more difficult for attackers to take down the application via DDoS attacks, insert malicious content using cross-site scripting attacks, or perform other common attacks. Voters can also be sure that they are using the correct version of the application because each file and directory in IPFS has its own unique hash. By verifying the hash of the application, voters can confirm that they are using the correct application and not a malicious clone.

Talking about the encryption of voter data, homomorphic encryption (see section 4.5) is also used here. This protects voter privacy and makes it easy to tally the votes and only decrypt the final tally. Zero-knowledge proofs (see section 4.6) are used to verify the correctness of encryptions and decryptions of votes.

For total transparency and auditability, the entire source code of both back-end and front-end are published on GitHub [30][31].

## 3.1 Delimitations

This project assumes the use of zero-knowledge proofs in its architecture. Without these proofs, the system is vulnerable to attacks from dishonest voters and a dishonest election authority. We couldn't find a different technology that would provide the necessary security to the voting system.

During the implementation, it became clear that it is currently very difficult to implement zero-knowledge proofs for this specific project for two reasons: encryption and decryption algorithms would have to be implemented in a low-level programming language with major limitations, and the method to construct proofs is currently inconvenient for users.

However, we do believe that it is possible and would satisfy the requirements for this project. If the tools to construct zero-knowledge proofs become more powerful and accessible, it is certainly possible to use them here.

A detailed explanation of how such an implementation could look like can be found in section 5.1.5.

# 4 Project Approach

The following section provides an overview of the proposed solution for this project. First of, the requirements are specified. This is divided into general requirements required for any i-Voting system, and then a list of requirements specific to this proposed solution.

After the requirements have been established, an overview of the high-level architecture is provided. Further details are given by UML and sequence diagrams for different interactions with the system.

## 4.1 Requirements

### 4.1.1 i-Voting requirements

Wang et al. [32] compiled a list of requirements for i-Voting systems. They argue that the literature hasn't agreed on a common set of requirements yet, so they went through a long list of publications and intersected all of the mentioned requirements. From this, they argue to have a complete list of all core requirements any i-Voting system needs to have. Additional requirements are included, which are favourable to have, but not strictly necessary. We will use both the core requirements and the additional requirements from their research in this project, as this is the most comprehensive list of requirements for i-Voting systems available today. Compiling our own list might produce an incorrect or incomplete list. With each requirement, we list a possible solution from what we have learned from the State of the Art.

Core Requirements:

1. **Correctness**. All votes are correctly counted. All valid votes need to be counted and any unauthorized or unauthenticated votes can not be counted.
    (a) *Possible Solution:* Ethereum Smart Contracts & Zero-knowledge proofs
2. **Privacy**. No one except the voter himself can know the voter's ballot choices. This is often paired with anonymity, which additionally means that no one can know if a voter has participated in the vote. This is not a requirement however.
    (a) *Possible Solution:* Homomorphic Encryption
3. **Unreusability**. A voter can not cast a vote twice.
    (a) *Possible Solution:* Ethereum Smart Contracts
4. **Eligibility**. Only authorized voters can vote.
    (a) *Possible Solution:* Ethereum Smart Contracts & MetaMask Ethereum Wallet

5. **Robustness**. The system can properly function with a number of misbehaved voters or partial failure of the system.
    (a) *Possible Solution:* Blockchain (back-end) and IPFS (front-end)
6. **Verifiable**. A voter can verify that his ballot has been counted.
    (a) *Possible Solution:* Ethereum Smart Contracts / Blockchain
7. **Usability**. A broad term that plays an important role in the success of the system.
    (a) *Possible Solution:* Best effort to develop an accessible solution. This requirement is not measurable in simple yes or no terms.

Additional Requirements:

8. **Fairness**. No partial results are computed before the end of the election.
    (a) *Possible Solution:* Ethereum Smart Contracts & Homomorphic Encryption
9. **Uncoercability**. Voters can not be forced to cast their vote in an unintended way. Votes can not be sold.
    (a) *Possible Solution:* Voters can change their vote during the election (see section 2.3.1.1)
10. **Efficiency**. The complexity of computation given the vast amount of voters.
    (a) *Possible Solution:* Encryption and Decryption is off-chain (client-side)
11. **Mobility**. Votes can be cast using mobile devices.
    (a) *Possible Solution:* A mobile-friendly web app and Ethereum wallet
12. **Vote-and-Go**. Voters can go offline after submitting their ballot. No further computation from the voter's side is needed.
    (a) *Possible Solution:* Ethereum Smart Contracts
13. **Universal Verifiability**. Anyone can verify the final vote count.
    (a) *Possible Solution:* Zero-knowledge proofs & Homomorphic Encryption
14. **E2E-Verifiable**. End-to-End Verifiable systems produce a receipt to the voter that convinces him of the correctness of his vote without revealing the choice of the vote on the receipt.
    (a) *Possible Solution:* Blockchain
15. **Practicality**. The system does not assume the security of any of the components. For example, some systems assume that there are only safe communication channels, or that voter's devices are free from malware. This is not allowed with this requirement.
    (a) *Possible Solution:* Blockchain

### 4.1.2 System-specific requirements

Besides the collection of requirements of Wang et. al [32], there might also be requirements that are more specific to the proposed system in this project. These are presented here.

1. **2048-bit keys**: When reviewing *BroncoVote* in the State of the Art (see section 2.3.2), we became aware of the necessity to use key sizes of at least 2048-bits for asymmetric encryption schemes. *BroncoVote* and a real-life example in Moscow showed how important this is. Since Ethereum only supports numbers of up to 256-bits, we have to find a way to work around that limit.

## 4.2 High-level architecture

As already described in the hypothesis (see section 3), this project consists of mainly two entities: the Ethereum blockchain and a front-end web application. A diagram of the high-level architecture of the project can be seen in Figure 3.

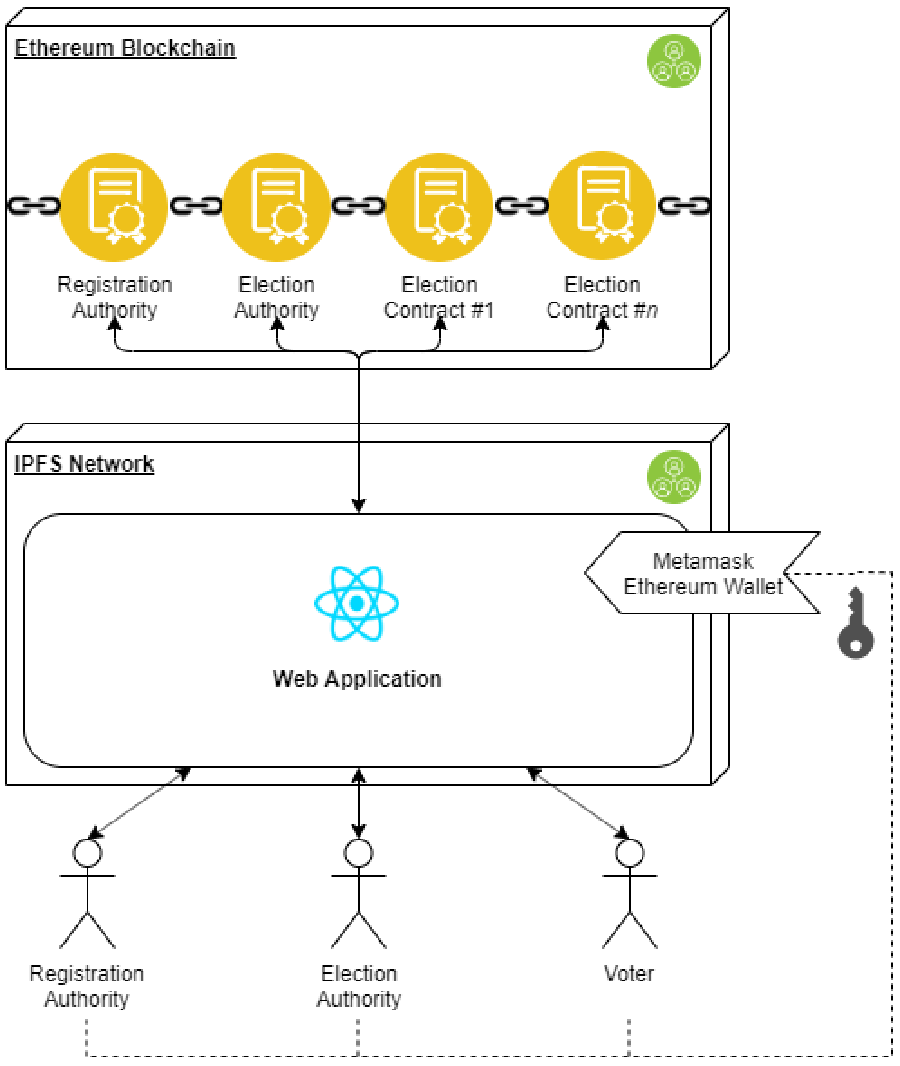


Figure 3: High-level System Architecture

### 4.2.1 Ethereum Blockchain

The Ethereum Blockchain is the most important part of the project. Three smart contracts (for an explanation of smart contracts, see 2.4.1.2) control the logic of the entire voting systems. They store a list of eligible voters, organizational details of every election, and the encrypted votes of each user in each election. Many of the requirements can be fulfilled by designing smart contracts in a secure way. A more detailed explanation of the purpose of each smart contract can be found in section 3.

### 4.2.2 Front-End Web Application

The front-end web application is the gateway for users to interact with the smart contracts on the blockchain. While it is possible to interact with smart contracts without a graphical interface, it is tedious and technical. A web application allows regular users to interact with smart contracts without even knowing what a blockchain is.

To achieve this, this project uses the open-source JavaScript library *React*, which is one of the most popular libraries used to develop interactive web applications in recent years.

### 4.2.3 Metamask and Web3

A question that did not come up in this report yet, but is very important, is user authentication. How can a smart contracts determine that the sender of a transaction is also authorised to perform the action? This works with *addresses*. Every account and smart contract on the network has a unique address. These are used to specify recipient and sender in transactions. An address can be mathematically derived from a private key, which a user keeps secret. In order to send a valid transaction, it has to be signed using that private key. So, if a user wants to participate in an election and send their ballot, they are required to have an Ethereum address.

In order to safely store a private key, users use so-called *wallets*. *Metamask* is currently the only sophisticated Ethereum wallet that runs in the browser as an extension. It allows users to generate a private key and Ethereum address, and most importantly injects a version of *web3* in the browser. Web3 is a JavaScript library that allows interaction with Ethereum nodes over HTTP connections. This allows users to use web applications that communicate directly with the Ethereum blockchain, without having to host an Ethereum node themselves.

Every time an application requests access to the users public address, Metamask asks the user for permission. The same goes for transactions. If an application wants to send a transaction to another address (e.g. a smart contract), the user is asked to confirm the transaction in a pop-up window.

### 4.2.4 IPFS

The InterPlanetary File System (see section 2.4.2) is used to host the front-end application. This eliminates many common attacks such as DDoS attacks and cross-site scripting attacks. It also allows users to be sure that they are using the correct application and not a malicious fake.

### 4.2.5 Users

There are three types of users that can use the web application. The *Registration Authority* is an independent authority that maintains a list of users that are eligible to vote. Each voter can have identifying information such as name, birth date and social security number that is tied to an Ethereum address. These addresses are then allowed to participate in elections.

The *Election Authority* is another authority that is responsible for creating elections and publishing the results afterwards. It is important to mention that the smart contracts forbid the authority to modify an election once it was created, and they are not able to see any votes during an active election.

The *Voter* can participate in elections by sending their encrypted ballots to the relevant election contracts.

## 4.3 UML Diagrams

Because of the fact that the main logic of this election system is based on Ethereum smart contracts, this is the main focus of the architecture. The front-end application may be far more complex to build, but it only provides an interface to interact with the smart contracts, which is why there is not much focus on the front-end here. Details about the front-end implementation are provided in Section 5.

UML diagrams are a good way to convey the structure of software components graphically. This is the first step in explaining the design of the three Ethereum smart contracts, and how they accomplish the necessary tasks and satisfy the requirements. A detailed explanation of the implementation of these smart contracts can be found in Section 5 as well.

The diagram in Figure 4 shows the attributes, methods, and restrictions of each smart contract, as well as the interaction between them. A useful feature of the programming language Solidity, which is used to write smart contracts, are function modifiers. These allow developers to specify certain requirements that have to be met before a function can be executed. In this project, many of the administrative functions such as registering a voter should only be allowed to be done by the qualified official. This is a problem, because the smart contracts are exposed to the entire Ethereum network and anyone could try to execute functions in the smart contract. To avoid this, many functions are tagged with a *manager* modifier, which specifies that only the creator of the smart contract is allowed to execute the function. And in the election contract, some functions need to be only accessible before, during, or after the election. For example, it shouldn't be possible to retrieve a user's vote while the election is still active, and it also shouldn't be possible to vote before the election started. These function modifiers can be found in the bottom section of each smart contract in the diagram. Each function is then marked with specific modifiers. Additionally, a *plus* means that the function or attribute is accessible to everyone, while *minus* means it is only accessible to the smart contract itself.

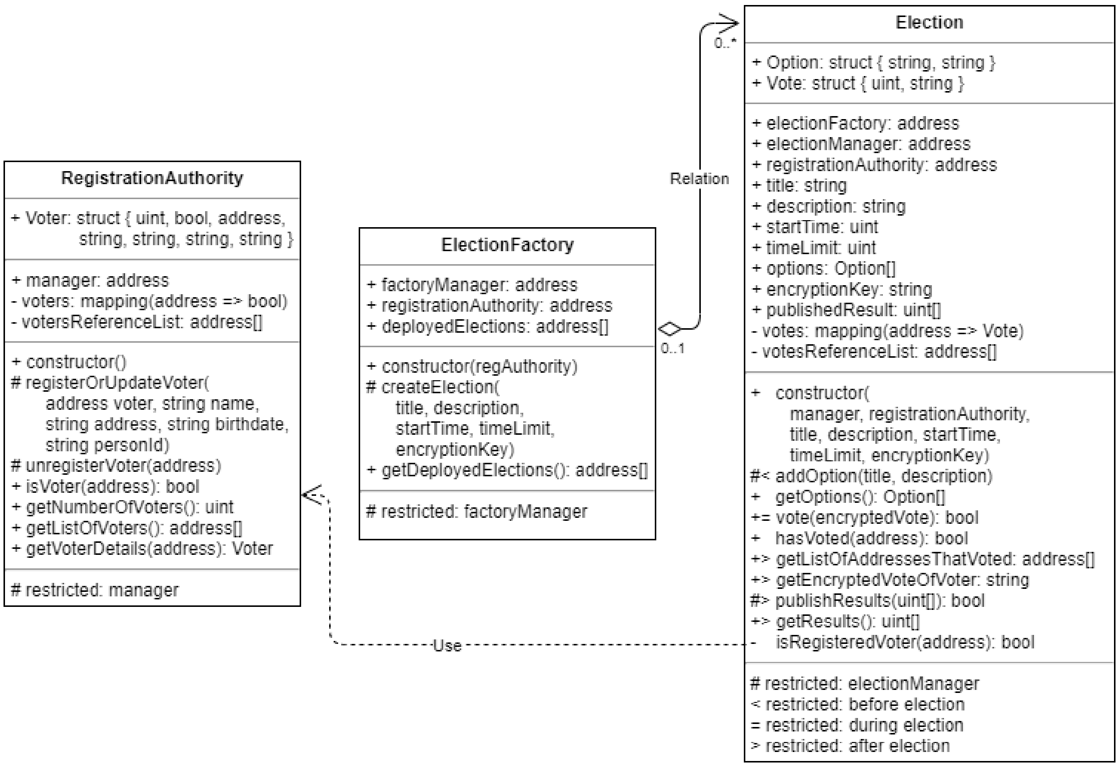

RegistrationAuthority
+ Voter: struct { uint, bool, address, string, string, string, string }
+ manager: address
- voters: mapping(address => bool)
- votersReferenceList: address[]
+ constructor()
# registerOrUpdateVoter( address voter, string name, string address, string birthdate, string personId)
# unregisterVoter(address)
+ isVoter(address): bool
+ getNumberOfVoters(): uint
+ getListOfVoters(): address[]
+ getVoterDetails(address): Voter
# restricted: manager
ElectionFactory
+ factoryManager: address
+ registrationAuthority: address
+ deployedElections: address[]
+ constructor(regAuthority)
# createElection( title, description, startTime, timeLimit, encryptionKey)
+ getDeployedElections(): address[]
# restricted: factoryManager
Relation
0..*
0..1
Use
Election
+ Option: struct { string, string }
+ Vote: struct { uint, string }
+ electionFactory: address
+ electionManager: address
+ registrationAuthority: address
+ title: string
+ description: string
+ startTime: uint
+ timeLimit: uint
+ options: Option[]
+ encryptionKey: string
+ publishedResult: uint[]
- votes: mapping(address => Vote)
- votesReferenceList: address[]
+ constructor( manager, registrationAuthority, title, description, startTime, timeLimit, encryptionKey)
#< addOption(title, description)
+ getOptions(): Option[]
+= vote(encryptedVote): bool
+ hasVoted(address): bool
+> getListOfAddressesThatVoted: address[]
+> getEncryptedVoteOfVoter: string
#> publishResults(uint[]): bool
+> getResults(): uint[]
- isRegisteredVoter(address): bool
# restricted: electionManager
< restricted: before election
= restricted: during election
> restricted: after election

Figure 4: UML Class Diagram of the Ethereum Smart Contracts

Additionally, the Election contract has two *structs*. Structs are custom data types that contain several other variables in one object. This project uses two of them, one to store a voting option (i.e. a candidate), and one to store an encrypted vote. An Option contains both a title and a description of a voting option, which could be a candidate name and his party affiliation.

The smart contracts are largely independent from another, and there are just two occasions where there is an interaction between them. The Election Factory contract keeps a list of addresses of Election contracts because it would otherwise be impossible to keep track of how many Election contracts there are. This is a one-to-many relationship as there is only one Election Factory, but there could be multiple Elections. The second case of interaction is a query by an Election contract to the Registration Authority to verify whether an address belongs to an eligible voter. This is used before a vote is recorded.

## 4.4 Sequence Diagrams

To understand the interactions that are shown in the high-level architecture and UML diagram, it is useful to present sequence diagrams. They make it easy to depict the information flow of applications and improve the understanding of the architecture.

Most interactions in the system are straight-forward and only consist of one step. Examples for this are registering a voter, retrieving information such as the results of an election, or adding candidates to an election before it has started.

The sequence diagrams in Figure 5 shows the steps involved in an entire election cycle, starting with registering voters, creating an Election contract, voting itself, and finally tallying and publishing the results. This involves several stages of the election and is designed to also show how the homomorphic encryption is used in this project to ensure voter anonymity.

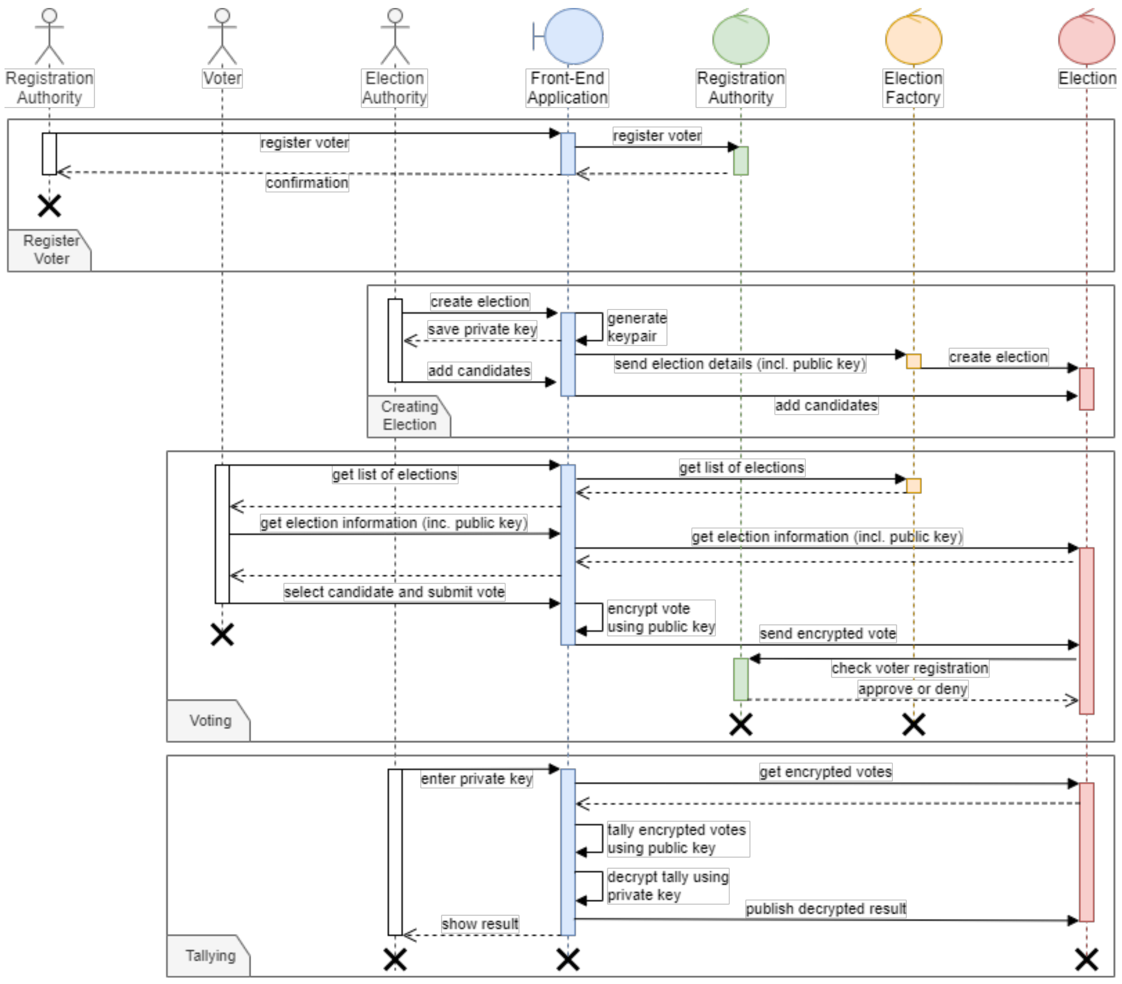


Figure 5: Sequence Diagram of the most common interactions in the system

## 4.5 Homomorphic Encryption

From the State of the Art, we have seen that the majority of proposed i-Voting schemes rely on the homomorphic encryption scheme, and this project is no exception. We have mentioned the term many times and delivered a short explanation of the benefits in the review of *BroncoVote* (see section 2.3.2). Since many of the requirements (see section 4.1) rely on homomorphic encryption, this section provides a more detailed overview of the scheme and how exactly it satisfies many of the requirements.

A cryptosystem is *homomorphic* if it has the property that the multiplication of two ciphertexts is the sum of the plaintexts associated with the ciphertexts.

Mathematically, that is $\varepsilon(m) \otimes \varepsilon(m') = \varepsilon(m \oplus m')$, where $\varepsilon$ is an encryption algorithm and $\otimes$ and $\oplus$ operations on the ciphertexts and plaintexts [33].

The advantage of this property is that the computation can be performed on encrypted data, and the private decryption key doesn't need to be known. This is useful in many areas, such as voting. Ron Rivest, one of the inventors of the RSA algorithm, created a video in collaboration with *Numberphile* about homomorphic encryption and how it can be used in voting systems [34]. He argues that a homomorphic voting system could be the future, as it is highly transparent and auditable. Making every encrypted vote publicly available allows anyone to verify that their vote has been counted, and that it has been counted correctly. The homomorphic property also allows for anyone to compute the final tally in an encrypted state, and the owner of the private key then only needs to convince everyone that this encrypted number results in a certain decrypted number. To do this, zero-knowledge proofs can be used (see section 4.6).

This philosophy translates well into this project, where homomorphic encryption is used to encrypt votes and tally the results. Since we are using a blockchain, each vote is also automatically publicly available, while still protecting the vote itself.

When a new election is created, a key-pair with private and public key is automatically generated. The public key is made available through the smart contract, and the private key is kept secret by the Election Authority. When a voter submits their choice, the front-end application retrieves the public key from the smart contract and encrypts the vote. The encrypted vote is then sent to the smart contract. When the election is over, the Election Authority can enter the private key in the application, which will automatically tally, decrypt, and publish the results of the election.

## 4.6 Zero-knowledge proofs

In addition to homomorphic encryption, zero-knowledge proofs are also used in this project. The only project reviewed in the State of the Art that implemented zero-knowledge proofs was *Hääl* (see section 2.3.3). This section provides an overview of the concept and how it can satisfy some of the requirements.

This project talked a lot about homomorphic encryption so far, and how it greatly improves voter privacy. But there is a fatal flaw: encryption and decryption of votes are computed client-side. The smart contracts only provide a storage space for the votes, they can not encrypt votes themselves. For one, this is because it is computationally expensive, and blockchains aren't made to perform complex calculations. Secondly, sending unencrypted votes to a smart contract would expose the vote to the public, as the blockchain is fully transparent.

In a real-life voting system, dishonesty has to be accounted for. If the entire source code of the project is publicly available, it is easy for an attacker to find the encryption methods. No one could stop the attacker from building a custom application that encrypts votes in a malicious way. For example, instead of encrypting a *1*, representing one vote for a candidate, the attacker could simply encrypt a *1000*. Since the result is encrypted and looks like every other vote, it is impossible to know that this vote is malicious. At the tally, the number of voters that participated are compared to the number of decrypted votes, so this type of attack would not go unnoticed. However, it would make the election invalid and damage the trust in the system.

The solution to this problem are zero-knowledge proofs. This is a method to prove a fact to another party without revealing the fact itself. The verifying party has zero knowledge of the fact itself, but can still be convinced that the proving party knows that fact. This is useful for this project, because it would allow voters to convince the smart contract that their encrypted vote follows the rules and is valid without revealing their vote. Furthermore, it can be used to prove that the published result of an election is the actual decryption of the encrypted tally.

Specifically, *non-interactive* zero-knowledge proofs don't even require an interaction between the proving and the verifying party. A common reference string shared between the party is sufficient to achieve consensus. *zk-SNARK* (zero-knowledge succinct non-interactive argument of knowledge) is a method that is widely used in the Ethereum world today, and would solve the problem of trusting voters to verify their votes correctly and the Election Authority to decrypt the results correctly.

A more detailed explanation of how zk-SNARKs could be implemented can be found in the *Project Execution* section (see section 5.1.5).

# 5 Project Execution

The implementation of the proposed solution was conducted in two steps: the *"back-end"* consisting of Ethereum smart contracts, and the front-end in form of a progressive web application. The smart contracts were implemented first as they define the core structure of the system and perform most of the critical logic. The methods exposed to external actors such as a web application can be considered similar to an API in a regular client-server architecture. The front-end provides an easy-to-use interface to the end users to interact with the smart contracts. While it does allow and prohibit different user groups to access certain sets of features (e.g. only eligible voters are allowed to cast a vote), this is only "soft" security. The smart contracts are exposed to the entire world and anyone is free to build their own application to interact with them. This means that all critical security measures are provided by the smart contracts. Due to this, and the fact that deployed contracts cannot be updated, these smart contracts are the part of the system that needs to be tested most thoroughly during the testing phase.

## 5.1 Ethereum Smart Contracts

As we already mentioned in the introduction to Ethereum (see section 2.4.1.2), the programming language used to develop Ethereum smart contracts is called *Solidity*. It is statically typed and object-oriented, with the syntax being influenced by C++, Python and JavaScript. Solidity code is compiled to bytecode that is executable by the *Ethereum Virtual Machine* (EVM). High-level abstractions such as inheritance, libraries, *mappings* and *structs* are supported. While the language has limitations (e.g. limited possibility of generating random numbers, time stamps being dependent on the miners in the network, etc.), it makes it considerably easier for developers to create smart contracts and deploy them on the various Ethereum networks.

Once a smart contract is deployed, it cannot be changed anymore. It is open to anyone on the network and possibly a target for malicious actors. Especially with smart contracts handling critical transactions such as a lottery, bank, or a voting system, it has to be assured that there are no vulnerabilities in the contract. There could be bugs in the contract, but also in the compiler or the blockchain itself. Because of this, smart contracts should be thoroughly tested and possibly audited. In this project, we created a suite of unit tests to test the various aspects of the contracts. This should eliminate the most obvious issues. If the contracts were to be used in actual elections, we believe that the contracts should be reviewed and audited by Ethereum experts and the community, which is why the code is open-source [30]. The testing of the smart contracts is described in Section 6.

The following subsections will explain the most important aspects of the implementation.

### 5.1.1 Registration Authority

The registration authority is responsible for keeping a list of all Ethereum addresses that are allowed to participate in elections. To prevent users from registering themselves without allowance, we decided to use an authentication model that doesn't require authentication over the Internet. It is difficult to implement an authentication model that can make sure that the person trying to register themselves is the actual person behind the computer. The case of Estonia's i-Voting system showed that researchers were able to create malware that could pretend to be a certain voter (see section 2.3.1.3). Concerns like this drove us to use a model where authentication is conducted offline. As an example, a user would go to the Registration Authority in the real world and show the relevant passport or ID card, and provide their Ethereum address. The authority verifies the voter's identity and records his Ethereum address. Additionally, identifying details such as name, address, or social security number can be stored in the smart contract in an encrypted form. Each voter can then easily verify if their Ethereum address is registered correctly, and the authority could manage the voter database. Because the Ethereum address is controlled by the voters themselves, in a case of a lost private key, or a case of death, the authority would be able to remove such addresses. With a model where voters register and unregister themselves, a stolen account couldn't be prohibited from voting.

The following code listing shows the instance variables used in the smart contract.

```
struct Voter {
    uint listPointer;
    bool isVoter;
    address ethAddress;
    string name;
    string streetAddress;
    string birthdate;
    string personId;
}

address public manager;
mapping(address => Voter) private voters;
address[] private votersReferenceList;
```

The *struct* is similar to a class in other programming languages. Objects of the struct can be instantiated and used. The *Voter* struct is used in a *mapping* from Ethereum addresses to instances of the structs. It is a key-value store that ties a voter to his or her address. In Solidity, every value in a mapping is instantiated by default. This means that an unrelated address would return an object with all values on their default values (`uint`'s on `0`, `bool`'s on `false`, `string`'s with an empty string). To avoid uncertainties, each Voter struct has a bool *isVoter* that is set to true if the address is actually registered. Each struct also contains a *list pointer*. Since the mapping returns an object for every key, it is impossible to tell how many voters there actually are, and what their addresses are. To keep track of that, a dynamic array is used to store all addresses that are registered as voters. The list pointer is then the index in that array. It is technically not necessary to have such an array, as an address can be looked up every time someone tries to vote. However, it is convenient to have a list if the front-end application wants to show an overview of registered addresses, and how many are registered.

The following code listing shows the constructor of the contract and a function modifier.

```
constructor() public {
    manager = msg.sender;
}

modifier restricted() {
    require(msg.sender == manager, "only the contract manager is allowed to use
        this function");
    _;
}
```

We have mentioned function modifiers before (see section 4.3). They ensure that certain conditions are met before executing a function. In this contract, we use one function modifier called *restricted*. It verifies whether the sender of a transaction is also the manager of the contract. If the sender is not the manager, the function will exit immediately. In the constructor, the sender (in this case the deployer of the contract) is set to be the contract manager. The function modifier is used to restrict access to registering and deregistering voters.

The following code listing shows the function responsible for registering a new voter, which can also be used to update information about an existing voter.

```
function registerOrUpdateVoter(address _voter, string _name, string _streetAddress
    , string _birthdate, string _personId) external restricted {

    if (voters[_voter].isVoter == false) {
        voters[_voter].listPointer = votersReferenceList.push(_voter) - 1;
        voters[_voter].isVoter = true;
        voters[_voter].ethAddress = _voter;
    }

    voters[_voter].name = _name;
    voters[_voter].streetAddress = _streetAddress;
    voters[_voter].birthdate = _birthdate;
    voters[_voter].personId = _personId;
}
```

The function is tagged with the function modifier *restricted*, as well as *external*. The external modifier is built into Solidity and means that only external transactions are accepted. The smart contract itself can not access it. The front-end application is such an external entity.

The function itself simply updates the struct instance in the mapping with the given parameters. The *isVoter* boolean is set to `true` if it wasn't already, and the address is added to the array of registered voters. A set of additional information can be added in the form of strings, but is not required. Since all data on the blockchain is public, it is advisable to encrypt additional information.

The function to deregister a voter is very similar and not listed here for the sake of keeping it short and interesting. All source code can be found on GitHub [30]. The function simply sets the *isVoter* boolean to `false` and leaves the rest of the data as is. Afterwards, the address is removed from the array.

The contract has four more functions used for checking whether an address belongs to a registered voter, getting the number of voters, getting a list of addresses of all voters, and getting detailed information about a specific voter.

### 5.1.2 Election Factory

The Election Factory creates new Election contracts and maintains a list of all elections. Such a contract is necessary if the system wants to support multiple elections at the same time. It also separates administrational logic from the actual election logic. Giving each election their own separate smart contract reduces the complexity significantly.

Besides the constructor, there are only two functions. One for creating a new election, and one for retrieving the list of all elections and their Ethereum addresses.

The following code listing shows the instance variables and constructor.

```
address public factoryManager;
address public registrationAuthority;
address[] public deployedElections;

constructor(address _registrationAuthority) public {
    factoryManager = msg.sender;
    registrationAuthority = _registrationAuthority;
}
```

Similarly to the previous contract, a reference to the manager of the contract is stored in the variable *factoryManager*. It is used to restrict access to creating new Election contracts. The constructor expects the address to the previously described Registration Authority, which is then passed to every new Election contract.

The following code listing shows the function used for creating a new Election contract.

```
function createElection(string memory _title, string memory _description, uint
    _startTime, uint _timeLimit, string _encryptionKey) public restricted {

    deployedElections.push(
        address(new Election(factoryManager, registrationAuthority, _title,
            _description, _startTime, _timeLimit, _encryptionKey))
    );
}
```

The function expects several parameters, including a title and description, a time frame of the election in seconds since the Unix epoch (01.01.1970), and the public encryption key for the homomorphic encryption. Inside the function, a new instance of an Election contract with all the given parameters is created and deployed, and the address of the deployed contract is stored in an array for future reference.

The second function in the smart contract simply returns that array of addresses.

### 5.1.3 Election

The Election contract is the heart of the systems. It is responsible for all critical logic that performs and regulates the election process. While it isn't more complex than the previously shown smart contracts, it is the most comprehensive in terms of number of variables and functions.

The contracts functionality is separated into three different stages of an election: before, during, and after. Most of the functions are outfitted with function modifiers that enforce the time constraints provided to the contract. For example, it should only be allowed to vote while the election is ongoing, and not before or after.

The following code listing shows the three function modifiers used to enforce the time constraints.

```
modifier beforeElection() {
    require(now < startTime, "only allowed before election");
    _;
}

modifier duringElection() {
    require(now > startTime && now < timeLimit, "only allowed during election");
    _;
}

modifier afterElection() {
    require(now > timeLimit, "only allowed after election");
    _;
}
```

The keyword `now` is used to get the current block timestamp. It can be influenced by miners if they try to delay the mining of a block, but only to a certain degree. The average block time of an Ethereum block is 15 seconds, so it is certainly not a value that can be used for calculations requiring the accurate time. However, as elections usually last a whole day or even longer, 15 seconds are not much of a concern in this case.

The contract has many instance variables such as the title, time constrains, the encryption key, and more. In order to store the available candidates, another `struct` is used. The following code listing shows the struct and how it is used.

```
struct Option {
    string title;
    string description;
}

Option[] public options;

function addOption(string _title, string _description) external manager
    beforeElection {
    options.push(Option({ title: _title, description: _description }));
}

function getOptions() external view returns(Option[]) {
    return options;
}
```

The struct consists of just two strings that store the name of the candidate and his party affiliation. All available options are then stored in an array of that struct. The function *addOption* is used to add a candidate to that list. The function modifiers are `external`, `manager`, and `beforeElection`. This means that only the external manager of the contract can use the function, and only before the election has started. Once the election has started, there can be no additional candidates added. The reason why the list of candidates is not provided during the creation of the contract is due to a limitation of Solidity. It is currently not possible to pass an array of a struct into a function. The function *getOptions* then returns the array of candidates.

The following code listing shows the relevant code for recording and storing votes.

```
struct Vote {
    uint listPointer;
    string encryptedVote
}

mapping(address => Vote) private votes;
address[] private votesReferenceList;

function vote(string _encryptedVote) external duringElection returns(bool) {
    require(isRegisteredVoter(msg.sender), "message sender is not a registered
        voter");

    votes[msg.sender].encryptedVote = _encryptedVote;

    if(!hasVoted(msg.sender)) {
        votes[msg.sender].listPointer = votesReferenceList.push(msg.sender) - 1;
    }

    return true;
}

function isRegisteredVoter(address _address) private view returns(bool) {
    RegistrationAuthority ra = RegistrationAuthority(registrationAuthority);
    return ra.isVoter(_address);
}
```

Similarly to the Registration Authority, all votes are stored in a mapping between the voters address and their vote. An array of addresses keeps a list of all the addresses that have submitted a vote. The vote itself is stored in a string due to the size of the encryption. Originally, we used `uint` arrays, but each `uint` can only store up to 256 bits of data, while strings have no size limitation. Because we use 2048-bit asymmetric encryption (see section 4.1.2), this is not sufficient. The voter submits their vote in the JSON format, with one encrypted number for each candidate. The plaintext of this number can either be a 0 or a 1, and there can only be one 1 in the vote. This way, the encrypted tally of each candidate can be calculated and decrypted.

The *vote* function requires the sender of the transaction to be a registered voter. To check this, the contract makes an internal call to the Registration Authority contract to look up a specific Ethereum address and return a result. If the address is verified to be registered, the encrypted vote is stored in the smart contract. The function allows voters to vote multiple times, with each vote overwriting their previous vote. We have learned from reviewing the Estonian i-Voting system that this could help prevent vote selling and voting under coercion.

The smart contract has many more functions that perform small tasks that we will not explain here. A list of all functions can be found in the UML diagram (see section 4.3) and in the source code [30]. They are mainly used to retrieve data from the contract, such as a list of addresses that voted in the election, the encrypted vote of a specific voter, or to check whether an address has already voted.

There is also a function that is used to publish the final results of the election in an unencrypted form. Once this is done, everyone can see the final result and verify that it is correct. To verify the election result, users can tally the encrypted votes for each candidate and verify the zero-knowledge proof provided by the Election Authority that proves that the published result is the actual decryption of that tally. The process of this is explained in section 5.1.5.

### 5.1.4 Compilation and Deployment

Ethereum smart contracts have to be compiled to bytecode, and the bytecode can then be used to deploy the contract to the blockchain. Many Ethereum projects use *Truffle*, a development framework for smart contracts, which makes testing, compiling, and deploying smart contracts easier. However, we chose to not use this tool for this project to not overcomplicate the process.

We developed two JavaScript scripts to compile and deploy the contracts to the blockchain. The compilation script uses *solc* [35], the official JavaScript bindings for the Solidity compiler. The compiler outputs the bytecode in the JSON format. The deploy script uses *Infuria* [36] to deploy the smart contracts. Infuria is an infrastructure service that allows easy access to the Ethereum network. Without such a service, one would have to run a full Ethereum node on their computer and then broadcast the new contract to their peers. This is complicated and time consuming. We use Infuria to deploy the smart contracts in this project. For demo and testing purposes, we are using the *Rinkeby Test Network* and not the main network, as the Ether on the test networks have no real value and can be freely experimented with.

Ethereum and Solidity is an ever-developing platform with breaking updates coming out regularly. Recently, version 0.5.0 of the Solidity programming language was released that introduced many breaking changes and a completely revamped way of compiling and deploying smart contracts [37]. At the time of developing the smart contracts, most of the available guides and tutorials were based on v0.4.x and it proved difficult to upgrade our code to the newest version. Especially the compile and deploy scripts radically changed and we couldn't integrate them with our existing way of compiling and deploying. Because of this, the smart contracts in this version are written for the last 0.4 version, which is v0.4.26. Ethereum is still under active development and those circumstances are not unusual. We don't have any concerns with using a slightly older version of the language, as it should provide the same level of security except for a few known bugs in the compiler.

### 5.1.5 Zero-knowledge proofs with ZoKrates

Zero-knowledge proofs are a method where a prover can prove to a verifier that he has knowledge of a fact, without revealing that fact to the verifier. We provided a quick introduction to this in section 4.6.

The logic behind this method is highly complex and we will not go into the details of how it works mathematically, as that would blow the scope of this project. Essentially, *zk-snarks* (Zero-Knowledge Succinct Non-Interactive Argument of Knowledge) have to satisfy three requirements [38]:

- **Completeness**: if the statement is true then a prover can convince a verifier
- **Soundness**: a dishonest prover can not convince a verifier of a false statement
- **Zero-knowledge**: the interaction only reveals if a statement is true and nothing else

To create such a proving system, arithmetic circuits are constructed. A prover can then run his secret parameters through the circuit and retrieve a proof that can be presented to the verifier. The proof doesn't contain the secret parameters. The logic of such circuits are generally quite simple and just verify that two values are equal, with only limited arithmetic operations available. The resulting arithmetic circuits, however, are highly complex and almost impossible to construct by hand.

Because of this, there have been several tools developed that aid in the creation of such circuits. In the area of Ethereum, a toolbox called *ZoKrates* [39] has gained popularity in the developer community. It provides a high-level programming language and a compiler for the arithmetic circuits. The tool can also generate a proving and a verification key. The verification key is then used to generate an Ethereum smart contract for verification purposes. Provers can then use their proving key to generate proofs with the help of ZoKrates. The proofs can be submitted to the smart contract, which then verifies the correctness of the proof. Such a smart contract could be used in this project to verify the correctness of votes and election results directly on the blockchain.

While ZoKrates is written in Rust, there is also a language binding available for JavaScript [40], making it possible to generate proofs in web applications. The toolbox is also available as an extension to Ethereum's official integrated development environment: Remix. This IDE runs in the browser and allows easy access to a Solidity compiler, debugger, deployer, and more. We used this tool extensively during the development of the smart contracts for debugging and testing purposes.

The following code listing shows an example proof written in the ZoKrates language, taken from the official documentation.

```
// This is an example from https://zokrates.github.io/gettingstarted.html
def main(private field a, field b) -> (field):
  field result = if a * a == b then 1 else 0 fi
  return result
```

This proof can verify that the product of a secret number equals a publicly known number. Input $a$ is private, meaning that it won't be visible in the proof. Input $b$ is the publicly known number. It is then checked whether the square of $a$ is equal to $b$. ZoKrates can compile this code into a smart contract, where a prover can submit proof that they have knowledge of a number $a$ that results in $b$ when squared, without revealing $a$.

As we stated in the delimitations (see section 3.1), we were unfortunately not able to implement the required zero-knowledge proofs for this project. However, it should certainly be possible if the tools to create them become more powerful.

Currently, the ZoKrates language is very limited. There are only two primitive data types, which are *fields* and *bools*. A *field* is essentially an unsigned integer with a maximum size of 254 bits. There is also only a limited number of standard functions available, requiring developers to write low-level code. When dealing with special encryption schemes and very large numbers, both of these facts become a major hurdle.

Implementing a custom version of the Paillier encryption scheme wouldn't be impossible, since the maths involved is fairly simple to compute. However, the limitations of variable size make this very difficult, if not impossible, at the current stage.

Theoretically, it should certainly be possible to implement the required proofs for this project, and it might even currently be possible with further research. However, this is out of the scope of this project.

To prove the correctness of decryption of the final tally, we think the following **pseudo**-code could potentially work.

```
def main(field decryptedResult, field encryptedResult, private field privateKey)
    -> (field):
    field result = if decryptedResult == decrypt(encryptedResult, privateKey) then
         1 else 0 fi
    return result
```

This proof would require the election authority to decrypt the final tally as normal on their local machine, and then submit the encrypted and decrypted result of the tally, as well as the private key. The computed proof will not include the private key. This proof would convince everyone that the decryption is accurate without revealing the private key. If we could implement a Paillier decryption method in ZoKrates, and have sufficient variable sizes, this is certainly possible. When the authority then submits the result to the Election smart contract, the contract could send the included proof to the Verifier smart contract first, before publishing the result.

Proving the correct encryption of votes is more complicated than that. Since the voter only has the public key for encryption, we can't prove the correct encryption. Every encryption of the same number with the same public key results in a different result with the Paillier encryption scheme, so we couldn't simply reproduce the encryption in the proof, as the private key is unavailable to a voter. Such a proof might require further methods that we are unaware of.

## 5.2 Front-End Application

The smart contracts, homomorphic encryption, and zero-knowledge proofs provide all the critical security measurements to satisfy the requirements for a secure i-Voting system, except *Usability* (requirement 7, see section 4.1) and *Mobility* (requirement 11, see section 4.1). These requirements can be satisfied with the fron-end application. A voting system can be highly secured, but if it doesn't provide an easy and accessible way of using it, it will not succeed. If voters don't understand what is going on behind the scenes, and don't understand the interfaces, they won't trust it. A voting system that isn't trusted is doomed to fail. Furthermore, the front-end application has to provide a secure way of authenticating voters that leaves no room for exploitation.

The authentication is provided by the *MetaMask* browser extension (see section 4.2.3). It is an Ethereum Wallet that allows web applications to interact with the Ethereum blockchain on behalf of the user. The private key is stored safely, and each transaction that the application is trying to make has to be confirmed by the user. It is the ideal solution for the authentication problem because it does not require the application to store any information about voters. In fact, everyone uses the exact same application without any personal configuration files, databases, or cookies.

The following section provides an overview of the interface design of the application, the technologies used to implement it, as well as the structure of the code and relevant code snippets.

### 5.2.1 Interface Design

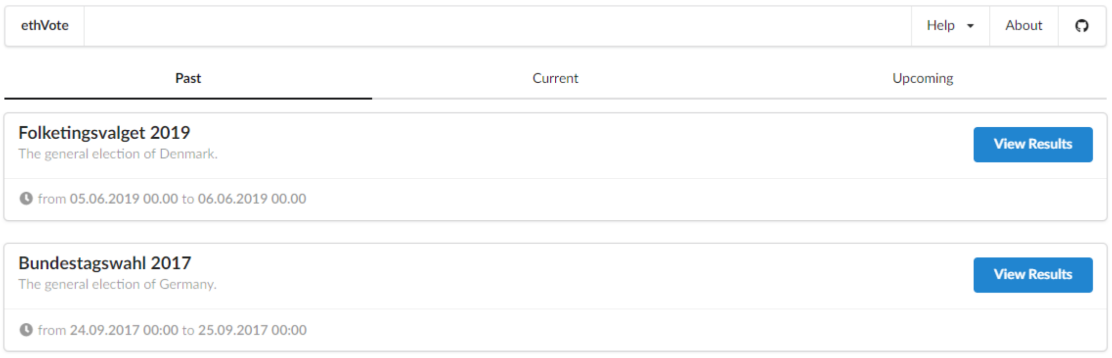
The goal of the interface is to be simple and informative. Users shouldn't have to require help when using it, as that might allow the manipulation of votes. Because of that, we wanted to design an application that is visually pleasing and not overwhelming. *BroncoVote*, a project that we reviewed in the State of the Art (see section 2.3.2), presented the entire application on one page and it was confusing to use in our review. With this project, we wanted to develop an application with multiple pages and conditional rendering to show relevant functionalities only to the users that are allowed to see it.

There are four types of users that will use the application: voters, non-voters, the election authority, and the registration authority. Based on the Ethereum address of the visiting user, which is obtained from MetaMask, functionality is shown or hidden.

The application detects if the visiting user has MetaMask installed, and redirects them to a specific help page if the extension is not installed. Any page that interacts with the Ethereum network will redirect users without MetaMask to that page, as it is required. Furthermore, if the wrong Ethereum network is selected in MetaMask, a separate help page has been set up where users are informed to select the correct network before continuing.

#### 5.2.1.1 Landing page

The landing page is similar for everyone. It shows a list of past, current, and upcoming elections in three tabs (see Figure 6). The list shows the title and description of the election, as well as its start and end time. A button on the right side allows users to open the election page. The button has a different text, colors, and icons depending on the circumstances. A menu at the top of every page allows users to go back to the main page, and access several help and information pages.

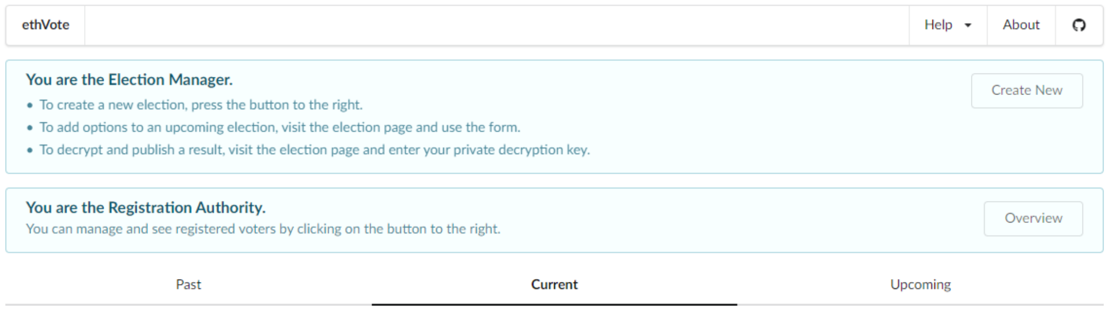


Folketingsvalget 2019
The general election of Denmark.
from 05.06.2019 00.00 to 06.06.2019 00.00

Bundestagswahl 2017
The general election of Germany.
from 24.09.2017 00:00 to 25.09.2017 00:00

Figure 6: Landing page and overview of elections

If the user is either the Election Authority or Registration Authority, an info box will be shown at the top that informs the user of that fact, describe what the user is allowed to do, and provide a button to the relevant page. This can be seen in Figure 7. In that case, the user is both authorities at the same time, which is just for demo purposes. Normally, they should be two different users.



You are the Election Manager.
- To create a new election, press the button to the right.
- To add options to an upcoming election, visit the election page and use the form.
- To decrypt and publish a result, visit the election page and enter your private decryption key.

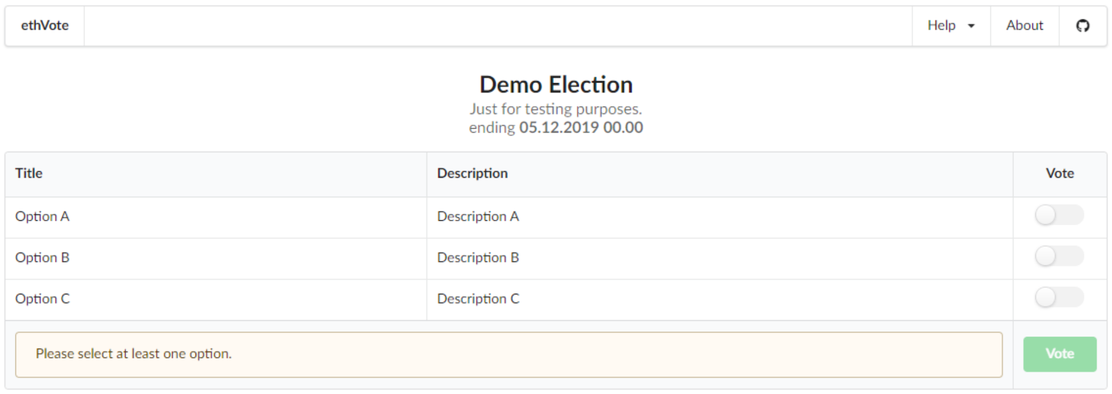
You are the Registration Authority.
You can manage and see registered voters by clicking on the button to the right.

Figure 7: Messages to the Election Authority and Registration Authority

#### 5.2.1.2 Election page

When clicking on a specific election, detailed information will load on a new page. Depending on whether the election is ongoing, already over, or starting soon, different lists will be shown.

Figure 8 shows how an ongoing election looks like to a registered voter. A list of candidates is shown, with both their title (name) and description (party), as well as a slider to select the option. The message in the bottom informs the user to select an option first, and enables the button if one was selected. If the user selects more than one option, the button is disabled again and a warning message shown.



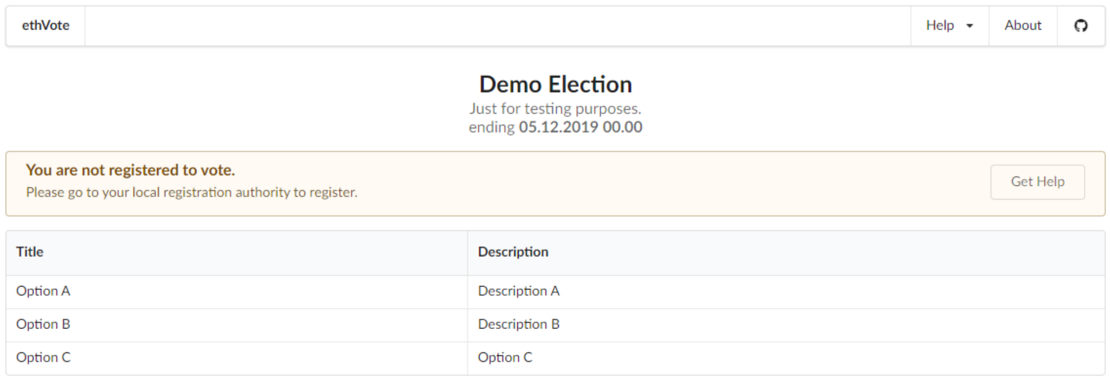
Demo Election
Just for testing purposes.
ending 05.12.2019 00.00

| Title | Description | Vote |
|---|---|---|
| Option A | Description A | |
| Option B | Description B | |
| Option C | Description C | |

Please select at least one option.

Figure 8: A voter can submit their vote in an ongoing election

If the user is not a registered voter, the options will still be shown, but without the interface to cast a vote. A warning message will be shown that informs the user that he has to be registered first. This can be seen in Figure 9.



Demo Election
Just for testing purposes.
ending 05.12.2019 00.00

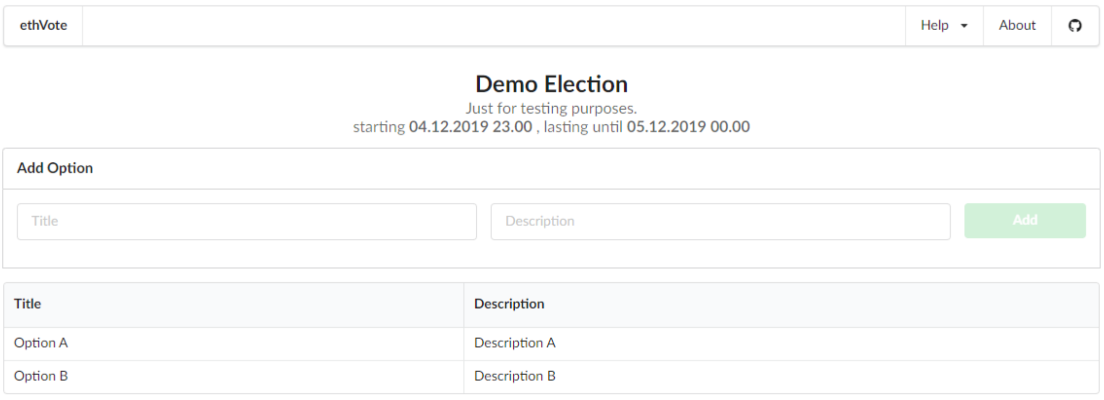
You are not registered to vote.
Please go to your local registration authority to register.

| Title | Description |
|---|---|
| Option A | Description A |
| Option B | Description B |
| Option C | Option C |

Figure 9: A voter can not vote if he is not registered

For an election that hasn't started just yet, the list will also simply show the candidates that will be available in the election. If the user is the Election Authority, an additional form on top of the list will be shown where candidates can be added until the election starts. An example of this can be seen in Figure 10.

For an election that is over, everyone will be able to see the results. However, until the votes have been tallied, decrypted, and published, the result will be shown as locked. Once again, the



**Demo Election**

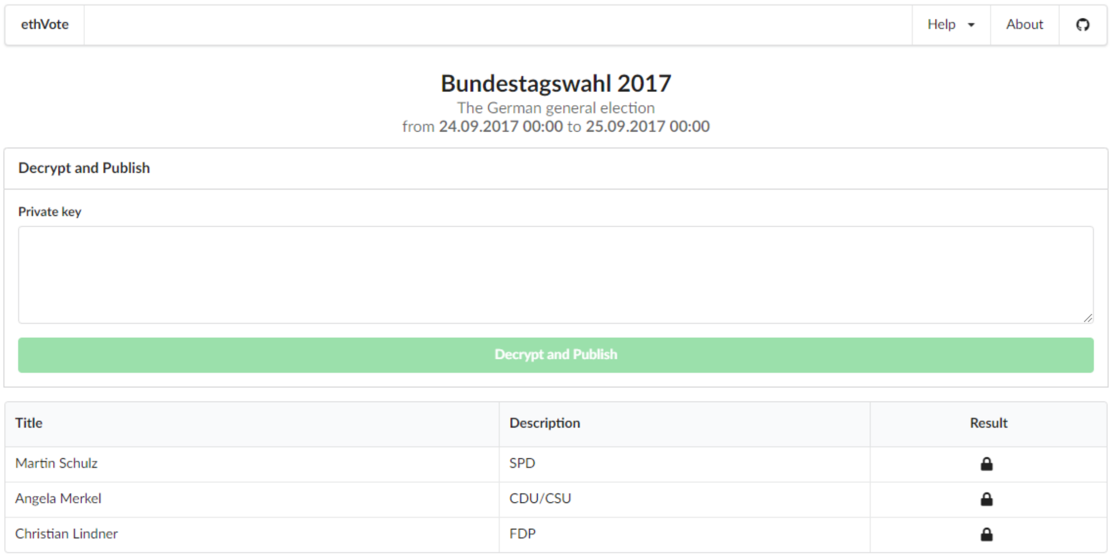
Just for testing purposes.

starting **04.12.2019 23.00** , lasting until **05.12.2019 00.00**

| Title | Description |
|---|---|
| Option A | Description A |
| Option B | Description B |

Figure 10: The Election Authority can add candidates before the election begins

Election Authority has an additional form at the top where the private key can be entered. The application will then automatically begin to fetch all votes, tally them, decrypt the tally, and publish the results. Afterwards, the lock is removed and everyone is able to see the final number fore each candidate. An example of this is shown in Figure 11.



**Bundestagswahl 2017**

The German general election

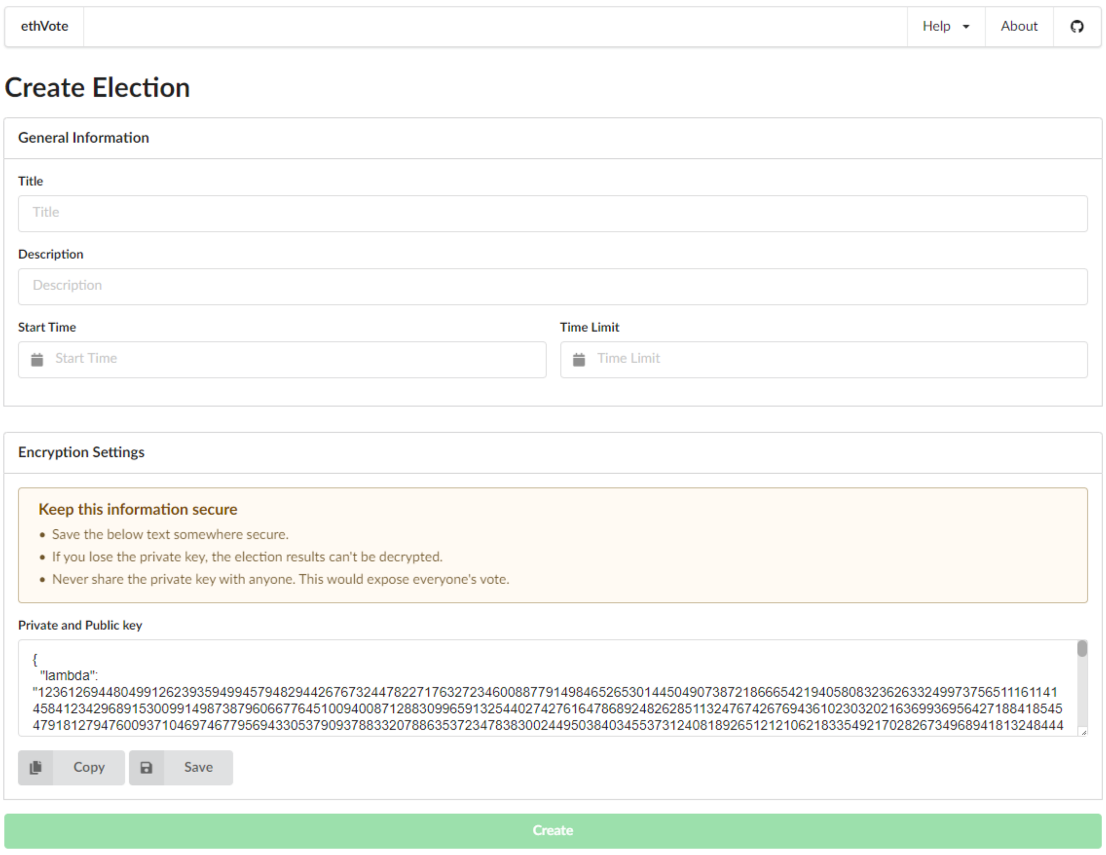
from **24.09.2017 00:00** to **25.09.2017 00:00**

| Title | Description | Result |
|---|---|---|
| Martin Schulz | SPD | 🔒 |
| Angela Merkel | CDU/CSU | 🔒 |
| Christian Lindner | FDP | 🔒 |

Figure 11: The Election Authority can tally, decrypt, and publish the election results

#### 5.2.1.3 Creating a new election

The page to create a new election is only accessible to the Election Authority. It consists of two form sections, where the user can enter details about the new election. A title and description is required, as well as the start and end date. When clicking on the date fields, a date and time picker will open where the user can select the desired date and time. The second part of the form shows the private and public key that will be used for the election. A new key-pair is generated every time the page loads, and the user is advised to save the key in a secure place. The public portion of the key is automatically published with the smart contract. An example of the page can be seen in Figure 12.



Keep this information secure

- Save the below text somewhere secure.
- If you lose the private key, the election results can't be decrypted.
- Never share the private key with anyone. This would expose everyone's vote.

Private and Public key

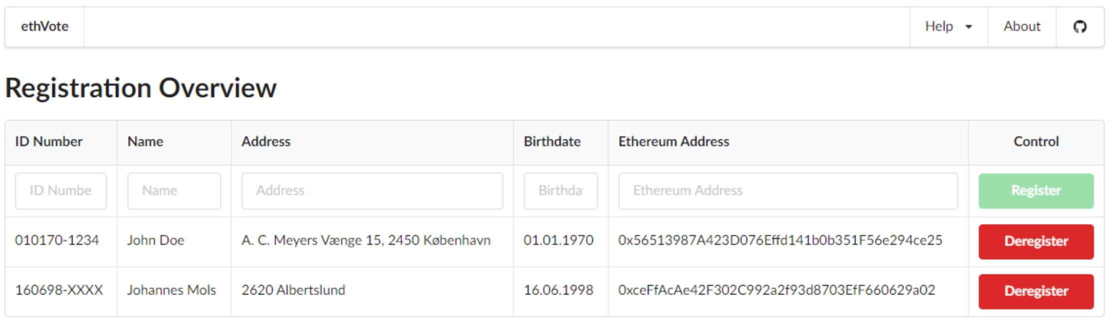
```
{
  "lambda":
"123612694480499126239359499457948294426767324478227176327234600887791498465265301445049073872186665421940580832362633249973756511161141
4584123429689153009914987387960667764510094008712883099659132544027427616478689248262851132476742676943610230320216369936956427188418545
4791812794760093710469746779569433053790937883320788635372347838300244950384034553731240818926512121062183354921702826734968941813248444
```

Figure 12: The Election Authority can create new elections

### 5.2.1.4 Managing registered voters

The page to manage the voter database is only accessible to the Registration Authority. A list of registered voters and their details are shown. As an example for this project, additional information about the voter can be added. This includes their name, ID number, address, and birthdate. The fields can be left empty. The most important detail about a voter is their Ethereum address, which is used to verify their eligibility to vote. A voter can be deregistered by clicking the red button next on the right side, and new voters can be registered by entering the necessary details and then clicking the green *"Register"* button. An example of this can be seen in Figure 13.

Since the personal information about voters is publicly stored on the blockchain, it would be wise to encrypt the data before submitting it. This is simple to do with a symmetric encryption scheme such as AES. The Registration Authority could choose a password and enter that to decrypt and show personal information, as well as to encrypt the information of new voters. However, we did not implement this in this project.



Registration Overview

| ID Number | Name | Address | Birthdate | Ethereum Address | Control |
|---|---|---|---|---|---|
| ID Numbe | Name | Address | Birthda | Ethereum Address | Register |
| 010170-1234 | John Doe | A. C. Meyers Vænge 15, 2450 København | 01.01.1970 | 0x56513987A423D076Effd141b0b351F56e294ce25 | Deregister |
| 160698-XXXX | Johannes Mols | 2620 Albertslund | 16.06.1998 | 0xceFfAcAe42F302C992a2f93d8703EfF660629a02 | Deregister |

Figure 13: The Registration Authority can register and deregister voters

### 5.2.2 Technologies

The front-end application is developed with the open-source library *React* (`https://reactjs.org/`). Together with *Angular* and *Vue.js*, it is one of the most used web frameworks currently available. The library allows developers to build complex user interfaces in a short span of time, including state management, routing, API interaction, and much more.

The reason why we chose to develop a web application instead of a desktop or mobile application is the cross-platform availability. Web applications don't need to be installed on a device, and they run on any device with a modern browser.

React applications can be extended with packages, which are additional libraries that can be used within JavaScript. React itself is also a package. The predominant package manager is *npm* (`https://www.npmjs.com/`). With over 1 million packages, the JavaScript ecosystem is highly active and offers about anything that a developer would need to develop any application.

Besides React, we use six additional packages in this project:

- **semantic-ui-react**: a UI framework with many components that are highly customizable.
- **semantic-ui-calendar-react**: additional calendar component that integrates well with the Semantic UI styling. We use this on the page to create new elections, where users can pick a date and time using this component.
- **react-router-dom**: allows routing to dynamic paths within the application.
- **file-saver**: allows users to save files to their hard drive.

- **paillier-js**: a JavaScript implementation of the Paillier cryptosystem, which is used in this project for its homomorphic properties.
- **web3**: Ethereum JavaScript API that allows the application to connect to Ethereum nodes and perform actions on the network. MetaMask injects an outdated version of the package into the application, which is why we use the latest version from the official package.

### 5.2.3 Implementation

The implementation involves a lot more code than the smart contracts, and it is largely irrelevant to the security aspects of this project. Because of this, we will only focus on a few interesting examples how the front-end application communicates with the Ethereum blockchain, and do not go into the implementation of the actual user interface. For the interested readers, the entire source code can be found on GitHub [31].

#### 5.2.3.1 Loading a Smart Contract

To communicate with a smart contract on the Ethereum blockchain, the application has to know what functions are available to be called. We explained that smart contracts are compiled to a JSON format (see section 5.1.4). This file contains both the bytecode and a so-called *Application Binary Interface* (ABI), which has information about the available functions. A new instance of a *web3* contract is created with the ABI and a given address. As we described above (see section 5.2.2), an outdated version of *web3* is injected into the application by MetaMask. To use the newest version of the library, we simply take the *provider* out of the injected version and initialize a new instance of the newer version with that provider. This can be seen in the code snippet below.

```
await window.web3.currentProvider.enable();
web3 = new Web3(window.web3.currentProvider);
```

A contract can then be loaded with the newly created *web3* instance. An example of this is shown below, where an Election contract is loaded given its ABI and Ethereum address, using the *web3* instance.

```
getElectionContract(web3, address) {
    const abi = JSON.parse(Election.interface);
    const contract = new web3.eth.Contract(abi, address);
    return contract;
}
```

#### 5.2.3.2 Retrieving data from smart contracts

Reading data from a smart contract is simple and straight forward. It does not cost any Ether, and the user does not have to confirm any transactions, meaning that the application can load data in the background and then show that data directly to the user after processing. This is one of the most common things that the application does, and it always follows the same pattern. In the code snippet below, the application creates an instance of the Registration Authority contract, and then checks whether the current user is a registered voter.

```
await window.web3.currentProvider.enable();
web3 = new Web3(window.web3.currentProvider);
regAuthority = this.getRegistrationAuthority(web3);
const userAddresses = await web3.eth.getAccounts();
const registered = await regAuthority.methods.isVoter(userAddresses[0]).call();
```

The methods of a smart contract can be accessed by calling `.methods.methodName()` on the contract instance, providing certain parameters if needed. The method can then be executed by calling `.call()` afterwards.

#### 5.2.3.3 Sending data to a smart contract

Sending data to a smart contract is very similar to retrieving data, except that modifying the state of a smart contract on the blockchain costs Ether. For every transaction that the application is trying to make, MetaMask will ask the user to confirm the transaction.

As an example, the code snippet below shows how a vote is submitted.

```
await this.props.contract.methods
    .vote(this.encryptVotes())
    .send({ from: this.props.userAddresses[0] });
```

The syntax is very similar to the one for retrieving data. However, the difference is that we have to call `.send()` instead of `.call()`. Because it costs Ether to submit data to the blockchain, the sender of the transaction has to be specified. In all cases, it is simply the current user of the application. MetaMask will then open a confirmation window to confirm the transaction, as can be seen in Figure 14.

The rest of the application communicates with the smart contracts in very similar ways, which is why we will not show more examples. The application essentially only retrieves and sends data from and to the relevant smart contracts using the shown methods.

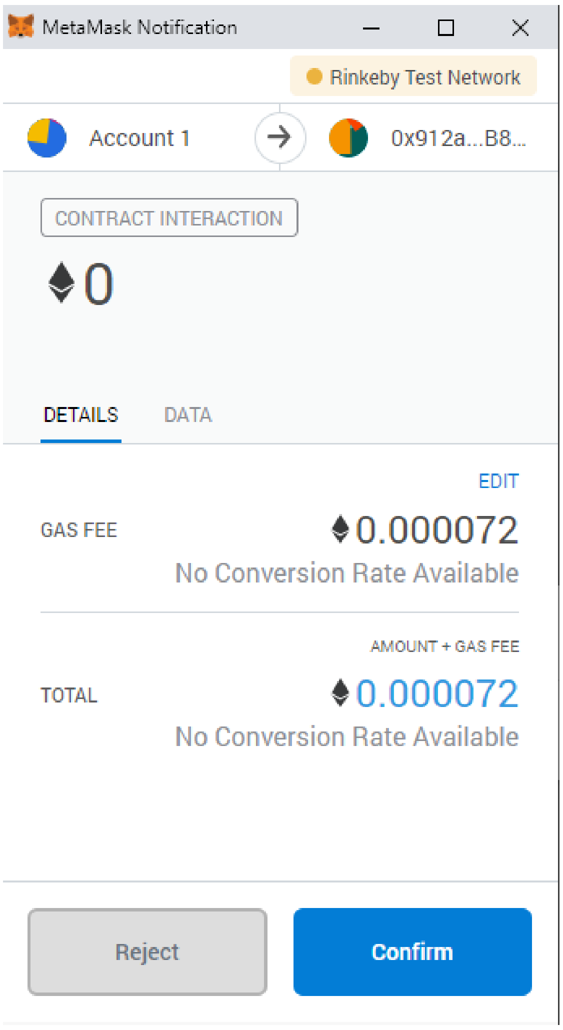


Figure 14: MetaMask confirmation window

# 6 Testing and Validation

This section will test and validate our proposed solution based on the specified requirements. Based on this, we can continue to argue that the proposed solution is a worthwile proof-of-concept that could be explored further and potentially be used in actual votes and elections.

## 6.1 Smart Contract Testing

Before the requirements can be validated however, we present our method of testing the correctness of the developed smart contracts. It is of utmost importance that these smart contracts do not contain design flaws, bugs, and errors. Once a smart contract is deployed, it can not be changed anymore. If someone attacks the smart contracts during the election, the election might have to be cancelled entirely, and that is assuming that the attack is noticed. To ensure the correctness of the smart contracts in this project, we wrote *unit tests* that test every aspect and functionality of the contracts. While this doesn't necessarily detect every error in the code or even compiler, it is a useful tool after making changes to a smart contract. If a change broke something unintentionally, the unit tests will report the error and it can be fixed.

### 6.1.1 Technologies

Conducting unit tests should always be done in a closed-off and controlled environment. This eliminates the risk of external factors playing a role in the tests. Such an environment can be produced by using a local Ethereum test network. One such tool is *Ganache* (`https://www.trufflesuite.com/ganache`). It is available as a Command-line interface (CLI) and NPM package (`https://www.npmjs.com/package/ganache-cli`), which allows us to use the local Ethereum network in a JavaScript environment. *Web3* is also used here, with the provider being Ganache instead of MetaMask as in the front-end application. Finally, writing unit tests in JavaScript is possible with the *Mocha* test framework (`https://mochajs.org/`).

### 6.1.2 Setup

The technologies described above are implemented in the following code snippet. Additionally, the code snippet shows a setup function that runs before every test. This function deploys a fresh copy of the Registration and Election Factory contracts, so that no previous tests have any influence on subsequent tests. Election contracts are always deployed during a test, and not before.

```
const assert = require("assert");
const ganache = require("ganache-cli");
const Web3 = require("web3");
const web3 = new Web3(ganache.provider());

const compiledRegistrationAuthority = require("../ethereum/build/RegistrationAuthority.json");
const compiledElectionFactory = require("../ethereum/build/ElectionFactory.json");
const compiledElection = require("../ethereum/build/Election.json");

const paillier = require("paillier-js");

let accounts, ra, ef;

beforeEach(async () => {
    accounts = await web3.eth.getAccounts();
    ra = await new web3.eth.Contract(
        JSON.parse(compiledRegistrationAuthority.interface)
    )
        .deploy({ data: "0x" + compiledRegistrationAuthority.bytecode })
        .send({ from: accounts[0], gas: "3000000" });
    ef = await new web3.eth.Contract(
        JSON.parse(compiledElectionFactory.interface)
    )
        .deploy({
            data: "0x" + compiledElectionFactory.bytecode,
            arguments: [ra.options.address]
        })
        .send({ from: accounts[0], gas: "3000000" });
});
```

### 6.1.3 Tests

Each smart contract has a set of unit tests that are designed to test every piece of functionality of the smart contract. In the following, we will list all of those tests. The implementation of each test can be found in the source code of the *ethVote* repository [30]. The code is very similar to the code we have shown for the implementation of the front-end application (see section 5.2.3), with the only difference being that this is a local test environment.

**Registration Authority:**

1. deploys the RA contract to the blockchain
2. the manager of the contract is also its creator
3. the number of registered voters is initialized with a zero
4. only the manager can access restricted functions
5. a voter can be registered
6. a voter can be unregistered

7. multiple voters can be registered and unregistered

**Election Factory:**

8. deploys the EF contract to the blockchain
9. the manager of the contract is also its creator
10. the contract has a valid address to the registration authority
11. the contract has zero deployed elections initially
12. only the factory manager can access restricted functions
13. an election contract can be deployed
14. multiple election contracts can be deployed

**Election:**

15. deploys an election contract to the blockchain
16. all constructor parameters are applied to the contract at creation
17. only the election manager can access restricted functions
18. the manager can add voting options before the election
19. a registered voter can cast their vote during their election
20. a non-registered voter can not cast their vote during their election
21. a voter can vote multiple times and invalidate their previous vote each time
22. votes are stored encrypted
23. election results can be published by the election manager

## 6.2 Requirement Verification

This project uses the list of requirements for i-Voting systems compiled by Wang et al [32] (see section 4.1). To verify whether the 15 requirements by Wang et al., as well as the one system-specific requirements, are fulfilled, we list each of them here again and explain how the proposed solution does or does not fulfill the requirement.

Fulfilled requirements are highlighted in green. Requirements that are not fulfilled in our implementation, but would be fulfilled with zero-knowledge proofs (see Delimitations, section 3.1), are highlighted in orange. And finally, unfulfilled requirements are highlighted in red.

**Internet Voting requirements:**

1. **Correctness.** The nature of the blockchain combined with the smart contract logic ensures that every vote is safe from manipulation and deletion. The vote itself will always be correct if it has been submitted correctly. The smart contract also enforces that only authorized voters can submit votes. However, without an implementation of zero-knowledge proofs, voters could submit fraudulent or invalid votes, which would invalidate the entire election due to the homomorphic tallying.

2. **Privacy.** It is impossible to know how a voter voted by just seeing the encrypted vote. The homomorphic encryption allows the tallying of votes without knowledge of the contents of the votes. Only the Election Authority could potentially use their private key to decrypt individual votes and not just the result. If the private key is leaked or misused, it could jeopardize voter privacy. This is a risk that is always present with encryption keys, and is not considered for this requirement.

3. **Unreusability.** The data structure used in the Election smart contract to store votes (see section 5.1.3) only allows one vote per Ethereum address, making it impossible to submit several votes as a single voter. Voters can however change their vote by overwriting their previous vote.

4. **Eligibility.** The Registration Authority smart contract keeps a list of all Ethereum addresses that are authorized to vote. It is not possible to submit a vote from an address that is not on that list.

5. **Robustness.** Both the smart contracts and the front-end application are using a peer-to-peer architecture, which makes it very robust to malicious actors to slow down or bring down a service. When it comes to malicious votes, this requirement can only be satisfied with a working zero-knowledge proof implementation. Otherwise, voters can submit fraudulent or invalid votes (see requirement 1, *Correctness*).

6. **Verifiable.** Voters can use a Blockchain explorer, MetaMask, or the front-end application itself to verify that the transaction has been executed successfully.

7. **Usability.** While it is difficult to verify this requirement in simple yes or no terms, we

do believe that our front-end application is easy and convenient to use. Only relevant information is shown, the layout is clean and simple, hints and warnings are provided at crucial points, and the wait for the confirmation of transactions is transparent with the help of loading spinners and time estimates.

8. **Fairness.** While partial results during the election are technically possible, there are obstacles to prevent it. The Election smart contract does not allow anyone to retrieve the encrypted votes before the election is over. However, it is possible for someone to monitor the transactions and manually retrieve the encrypted votes with the help of a block explorer such as `https://etherscan.io/`. Actual results from this can only be computed with knowledge of the private key, which the Election Authority keeps secret. If the Election Authority itself makes the effort to monitor transactions and tally them, partial results are indeed possible. It should however be in the authority's best interest to not do this, as it would undermine their credibility and trust from voters. Since it is only possible to compute partial results with the private key, we consider this requirement to be fulfilled.

9. **Uncoercability.** Internet Voting brings the risk of vote selling and voting under coercion. It is not possible to control what happens in the voters home, in contrast to a classical voting booth where only the voter is allowed inside. To avoid voting under coercion and vote selling, the system allows voters to cast their vote again while the election is still active. Voters could simply change their vote when they are no longer under coercion, or after selling their vote. This approach was taken from Estonia's i-Voting system (see section 2.3.1.1).

10. **Efficiency.** Performing computationally intensive work on a blockchain is expensive. Every calculation that a function of a smart contract performs costs the sender of the transaction gas (Ether). The more complex the calculation, the more expensive it is. Because of this, we do not use the smart contracts to encrypt or decrypt any information. They only serve as a data storage. The computationally intensive homomorphic encryption and zero-knowledge proofs are performed by the clients in the front-end application.

11. **Mobility.** MetaMask currently has a Beta version of their mobile application available for iOS and Android. It allows users to manage their Ethereum wallet on their phone, and use Ethereum-powered websites, including our front-end application. The application has a mobile-friendly layout, allowing voters to cast their vote on their mobile phone.

12. **Vote-and-Go.** Voters do not need to stay online after voting. Submitting their encrypted vote to the Election contract is sufficient.

13. **Universal Verifiability.** The homomorphic encryption scheme allows the computation of the encrypted final tally without the need for a private key. The Election Authority can then use the private key to decrypt the tally and publish the result. Since everyone is able to calculate the encrypted final tally themselves, the authority only has to prove

the correctness of that one decryption. With an implementation of zero-knowledge proofs as described in section 5.1.5, the Election Authority could convince everyone that the decryption is correct.

14. **E2E-Verifiable.** An End-to-End Verifiable system produces a receipt to the voter that convinces him of the correctness of his vote without revealing the choice of the vote on the receipt. While voters using our front-end application can be sure of the correctness of their vote due to the code being publicly available to verify, it might not be enough to convince everyone. Once again, an implementation of zero-knowledge proofs for the correctness of votes would satisfy this requirement. If the encrypted vote passes the zero-knowledge proof, the voter can be absolutely certain that his vote is correct, without his vote being revealed in the proof.

15. **Practicality.** A practicable system does not rely on assumptions that are unrealistic in large-scale scenarios. For example, we have seen from *BroncoVote* (see section 2.3.2) that they use an encryption server and assume the security of that server. The proposed solution does not have any components that are unrealistic to be secure in large-scale scenarios. The Ethereum network is already very large and withstood many attacks. Scalability is a problem that could be a bottleneck if thousands of voters try to vote at the same time. However, this is an issue that is actively being worked on and addressed in the upcoming Ethereum 2.0 release, where a Proof-of-stake consensus mechanism and Sharding will be implemented [41]. The more candidates are available in an election, and the more voters participate in it, the more data will have to be stored on the blockchain. This costs Ether and becomes expensive quickly with the current Ethereum price (1 ETH = €136 as of 08.12.2019). If the elections are conducted on the test networks or a private Ethereum network, the Ether has no value and only the electrical costs for computation matter. Furthermore, the front-end application is hosted on the peer-to-peer IPFS network. If the number of users of the network and application increases, the availability also increases. We also don't see a scalability problem with the encryption and decryption of votes. Every voter encrypts their vote themselves, and no encryption or decryption is computed on the blockchain. At the end of an election, the votes have to be tallied by the Election Authority, which could require a considerable amount of computational power depending on the number of candidates and voters. However, this is a one-time calculation that is not too expensive to calculate. Overall, the choice of technologies allows us to not rely on components that are not safe to be used in large-scale scenarios.

**System-specific requirements:**

1. **2048-bit keys.** We ended up circumventing the 256-bit integer limit in Solidity by storing the encryption key and votes as strings in JSON format. There is no size limit on strings in Solidity, and the only limit is gas cost of transmitting the data. Since the votes are only stored on the blockchain, and no calculations are performed directly on-chain, this is not a problem.

# 7 Discussion

In this report, we have presented a proof-of-concept i-Voting system that is supposedly secure. In the previous section, we have shown that all the requirements for such a system have been met, considering the delimitation of zero-knowledge proofs (see section 6.2).

However, this does not mean that the proposed solution can be used without concerns. In this section, we will discuss some of the things that can be an issue and what can be done.

## 7.1 Private Key Security

A major concern we have about the proposed solution is the trust that is given to the voters to keep their private keys to their Ethereum wallets secret. The blockchain is still a new technology and it is safe to assume that most of the voters that would use the system either do not know what a blockchain is or how it works, and have likely never used one knowingly. Considering that 61% of Internet users still reuse passwords on multiple sites [42], this is a problem to take seriously. In the State of the Art (see section 2.4.1.3), we explained that blockchain users can be vulnerable to phishing and dictionary attacks. If a voters private key is stolen, his vote could also be stolen. A possible solution to this is to educate voters before election how to handle private keys, and set up a system where voters can report their accounts to be lost or stolen so that the Ethereum address can be deregistered from the Registration Authority smart contract.

## 7.2 Gas Costs

Sending data to a smart contract can become expensive quickly. The user has to pay a fee (called gas) depending on the computational complexity of the transaction, and data has an influence on that. As of December 2019, Ethereum has a block gas limit of 10 million, which equals to 0.01 Ether. The block size is on average between 20 and 30 kilobytes [43].

This project uses a 2048-bit asynchronous encryption scheme to encrypt votes. Each candidate in the list gets their own encrypted vote that is either a 0 or 1, signalling whether the user voted for the candidate or not. That means that the more candidates there are, the larger the size of the vote becomes. Using the Paillier encryption scheme, an encryption using a 2048-bit key is 1233 characters long, equalling the same amount of bytes using ASCII encoding. Given that a block in Ethereum is currently normally not larger than 30 kilobytes, we can estimate that we can have a maximum of 24 candidates per election. Each vote would then take up an entire block and use almost the entire gas limit of that block. The cost and processing time of such a transaction is also considerable, especially when many voters try to vote at the same time. After testing this on the Rinkeby test network, it turned out that the maximum number of candidates is actually below 20, but in any case more than 10.

One possible solution to this is to use a private Ethereum network, where block and gas limits can be increased. This involves risks because the network will be considerably smaller, which makes it easier to attack (see section 2.4.1.3). In a national election however, this might be a good choice if voters are allowed to contribute as miners.

Another cost consideration is the fact that voters have to pay fees to sign the transactions they make when voting. Since the system would likely not run on the main network, but instead on one of the test networks or a private network, the Ether would have no actual value. However, users still have to obtain some Ether before they can sign any transactions. A solution to this could be to set up an Ether fountain where voters can request a certain amount of Ether. A more user-friendly solution could be to send out a fixed amount of Ether to every registered voter when a new Election contract is created. It is unfortunately not possible to pay the transaction fees as the receiver, which would be a more user-friendly solution.

## 7.3 Experimental Technology

Almost all technologies used in the proposed solution are still relatively new and often under active development. They can change or be abandoned at any time, and the security of the technologies is not necessarily guaranteed. While some of the technologies such as Ethereum and MetaMask have been audited, new developments can always open up new vulnerabilities. Ethereum smart contracts are also constantly changing and there is a list of known bugs in the Solidity compiler. The implementation of the Paillier encryption scheme used in this project is also experimental and has not been updated in over two years. It would also be necessary to fully implement zero-knowledge proofs, which we were not able to do in this project (see section 3.1).

If this solution were to be used in large-scale elections, all components should be thoroughly audited to ensure that there are no vulnerabilities. Given the previously described state of many of the technologies, we would recommend to wait a few years until they have matured enough to be used without concerns.

# 8 Conclusion

This project focused on current election systems and their security. An overview of the State of the Art revealed that paper ballots are still superior to Electronic Voting Machines and Internet Voting in terms of security, and EVM's should not be used at all. However, the cost of conducting paper elections beg the question if it is possible to create a digital voting system that could match the security of paper ballots for a cheaper cost. This led to the following research question.

*Is it possible to design an Internet voting system that can match the security of paper ballots? What are the requirements for such a system and how can they be fulfilled in a scalable and practical way?*

—Research Question, Section 1.1

To answer this question, we reviewed four different i-Voting systems. The first system is used in Estonia's national elections and was revealed to have vulnerabilities in operational and procedural security, making it not viable to be a solution. The three other projects all work with the Ethereum blockchain and seem to come much closer to being absolutely secure. However, we found flaws with all three projects that disqualify them from being a solution, although much closer.

The proposed solution in this project is similar in terms of the technologies and methods used in those three projects. It uses three Ethereum smart contracts to store votes and manage the security of the election (only allowing registered voters to vote, not allowing results to be retrieved before the election is over, etc.). The Paillier homomorphic encryption scheme is used to encrypt votes before submitting them to the blockchain, which protects voter privacy and allows anyone to compute the final tally in an encrypted form without knowing the private key. And finally, zero-knowledge proofs are used to prove to voters that the decrypted result is correct, as well as for voters to prove that their encrypted vote is valid. On top of that, we built a multi-page front-end application with React that makes it easy for voters and authorities to interact with the blockchain.

The requirements for i-Voting systems were acquired by Wang et al. [32], who compiled a list of the most frequently used requirements for such systems in current literature. There is a total of 15 requirements that mainly focus on the correctness and verifiability of the solution, while also providing voter privacy. The proposed solution satisfies all of these requirements in theory, but the implementation failed to implement zero-knowledge proofs due the lack of functionality from available tools (see section 3.1). However, we believe that it is possible to implement the proofs with further research.

During testing and the discussion (section 6 and 7), we realized that the solution does not scale well due to the current limitations of the Ethereum network. The size of encrypted votes becomes a problem when there are more than a dozen candidates, and the transaction speed of the network is currently not fast enough (approx. 15 transactions per second) to accommodate for thousands of voters voting at the same time. The scalability issues will be improved and hopefully resolved with the release of Ethereum 2.0.

Another important factor of an election system is the practicality of it. While we made it as easy as it gets with an interactive and easy-to-use front-end application, voters will still have to install an Ethereum wallet in form of a browser extension and generate an Ethereum address, as well as to save their private key / seed phrase. This is not a concern for users who are familiar with the technology, but will certainly be a problem if the system was adopted in a large scale.

To conclude, while there are concerns about scalability and practicality, we believe that the proposed solution is secure, and can in fact match the security of paper ballots. It satisfies all requirements, and unless there are vulnerabilities in the technologies that were used or in the implementation of the smart contracts, it should not be possible to compromise the election.

Does this mean that we believe it should be used in national elections? No, absolutely not. A voting system that wants to be successful does not only need to be secure, but it has to be trusted by voters. If there is no trust in the system, the results are likely not going to be accepted by everyone. The problem with i-Voting solutions is the complexity of their architecture and methods used. The average voter will not know what a blockchain is or how it works, or how homomorphic encryption ensures their privacy, or how a result can be trusted with zero-knowledge proofs. A paper ballot is much easier for a voter to understand and trust.

We are carefully optimistic that the technologies and their understanding in the population will mature in the coming decades which would allow for a system that is secure, scalable, and practical, which is also trusted by the voters.

# References

[1] Eulau Heinz, Roger Gibbins, and Paul David Webb. *Election.* 2015. URL: https://www.britannica.com/topic/election-political-science#ref229014.

[2] The Economist Intelligence Unit. *Democracy Index 2018.* Tech. rep. 2018. URL: https://www.eiu.com/topic/democracy-index.

[3] Norbert Kersting and Harald Baldersheim. *Electronic Voting and Democracy: A Comparative Analysis.* Palgrave Macmillan Ltd, 2004. ISBN: 1–4039–3678–1.

[4] Estonia. *i-voting.* URL: https://e-estonia.com/solutions/e-governance/i-voting/.

[5] Drew Springall et al. "Security Analysis of the Estonian Internet Voting System". In: *Proceedings of the ACM Conference on Computer and Communications Security.* 2014. ISBN: 9781450329576. DOI: 10.1145/2660267.2660315. URL: https://estoniaevoting.org/.

[6] David Evans and Nathanael Paul. "Election Security: Perception and Reality". In: *IEEE* (2004), p. 8. DOI: 10.1109/MSECP.2004.1264850. URL: https://ieeexplore.ieee.org/document/1264850/.

[7] SPIEGEL Online. *Kosten für Bundestagswahl so hoch wie nie.* 2017. URL: https://www.spiegel.de/politik/deutschland/bundestagswahl-kosten-laut-innenministerium-hoch-wie-nie-a-1164713.html.

[8] Scott Wolchok et al. "Security analysis of India's electronic voting machines". In: *Proceedings of the ACM Conference on Computer and Communications Security.* 2010. ISBN: 9781450302449. DOI: 10.1145/1866307.1866309.

[9] Thomas W Lauer. "The Risk of e-Voting". In: *Electronic Journal of e-Government* (2004). ISSN: 01933892. URL: https://www.researchgate.net/publication/228920801_The_Risk_of_eVoting.

[10] Brennan Center for Justice. *Voting System Security and Reliability Risks.* Tech. rep. New York, 2016. URL: https://www.brennancenter.org/sites/default/files/analysis/Fact_Sheet_Voting_System_Security.pdf.

[11] Matt Blaze et al. "DEFCON 27 Voting Machine Hacking Village". In: (2019). URL: https://media.defcon.org/DEF%20CON%2027/voting-village-report-defcon27.pdf.

[12] Tom Scott. *Why Electronic Voting is a BAD Idea.* 2014. URL: https://youtu.be/w3_0x6oaDmI.

[13] Epp Maaten. "Towards remote e-voting: Estonian case". In: *Conference: Electronic Voting in Europe - Technology, Law, Politics and Society* (2004). URL: https://pdfs.semanticscholar.org/ff4d/0a77e7561e62fd0258280c0baa02d8256a03.pdf.

[14] "BroncoVote: Secure Voting System using Ethereum's Blockchain". In: *Proceedings of the 4th International Conference on Information Systems Security and Privacy (ICISSP)* (2018). DOI: 10.5220/0006609700960107.

[15] Wikipedia. *Key Size.* URL: https://en.wikipedia.org/wiki/Key_size#Asymmetric_algorithm_key_lengths.

[16] Pierrick Gaudry. *Breaking the encryption scheme of the Moscow internet voting system.* Tech. rep. 2019, p. 6. URL: https://arxiv.org/abs/1908.05127.

[17] Edilson Junior, Osorio. *Anonymous Electronic Voting System on Public Blockchains.* 2018. URL: https://github.com/eddieoz/haal.

[18] Johannes Mols. *Adding voters is unprotected?* URL: https://github.com/eddieoz/haal/issues/5.

[19] Polys. *Polys - Online Voting System.* URL: https://polys.me/assets/docs/Polys_whitepaper.pdf.

[20] Polys. *Is Polys open-source software?* URL: https://docs.polys.me/en/articles/1641286-is-polys-open-source-software.

[21] Polys. *Polys GitHub.* URL: https://github.com/polys-vote/polys-protocol.

[22] Polys. *What's the difference between Polys and Google Forms?* URL: https://docs.polys.me/en/articles/1641300-what-s-the-difference-between-polys-and-google-forms.

[23] Investopedia. *51% Attack.* 2019. URL: https://www.investopedia.com/terms/1/51-attack.asp.

[24] David Gilbert. *Russian cybersecurity firm Kaspersky wants to run your next election.* 2017. URL: https://www.vice.com/en_us/article/5955k3/kaspersky-polys-election-voting.

[25] Satoshi Nakamoto. *Bitcoin: A Peer-to-Peer Electronic Cash System.* 2008.

[26] Christian Stoll, Lena Klaaßen, and Ulrich Gallersdörfer. "The Carbon Footprint of Bitcoin". In: *Joule* (2019). ISSN: 25424351. DOI: 10.1016/j.joule.2019.05.012.

[27] Tobias Fertig and Andreas Schütz. *Blockchain für Entwickler.* 1st ed. Bonn: Rheinwerk Verlag, 2019, p. 565. ISBN: 978-3-8362-6390-0. URL: https://www.rheinwerk-verlag.de/blockchain-fur-entwickler_4677/.

[28] Parity. *Parity bug.* 2017. URL: https://www.parity.io/security-alert-2/.

[29] zkCapital. *IPFS Analysis.* Tech. rep. 2018.

[30] Johannes Mols. *ethVote.* 2019. URL: https://github.com/johannesmols/ethVote.

[31] Johannes Mols. *ethVote-react.* 2019. URL: https://github.com/johannesmols/ethVote-react.

[32] King-Hang Wang et al. "A Review of Contemporary E-voting: Requirements, Technology, Systems and Usability". In: *Ubiquitous International* (2017). ISSN: 2520-4165.

[33] Nina Pettersen. "Applications of Paillier's Cryptosystem". Master thesis. Norwegian University of Science and Technology, 2016, p. 152. URL: https://pdfs.semanticscholar.org/b3c9/5d839a48418e3674380a76b8fe6c960d37c7.pdf.

[34] Ron Rivest. *Was YOUR vote counted? (feat. homomorphic encryption).* 2016. URL: https://youtu.be/BYRTvoZ3Rho.

[35] Ethereum. *solc.* 2019. URL: https://www.npmjs.com/package/solc.

[36] *Infuria.* 2019. URL: https://infura.io/.

[37] Solidity Docs. *Solidity v0.5.0 Breaking Changes.* 2019. URL: https://solidity.readthedocs.io/en/v0.5.13/050-breaking-changes.html.

[38] Christian Reitwiessner. “zkSNARKs in a Nutshell”. In: *Ethereum Blog* (2016). URL: `https://blog.ethereum.org/2016/12/05/zksnarks-in-a-nutshell/`.

[39] ZoKrates. *ZoKrates: A toolbox for zkSNARKs on Ethereum.* 2019. URL: `https://github.com/Zokrates/ZoKrates`.

[40] ZoKrates. *ZoKrates-JS: JavaScript bindings for ZoKrates project.* 2019. URL: `https://github.com/Zokrates/zokrates-js`.

[41] Brian Curran. *What is Sharding? Guide to this Ethereum Scaling Concept Explained.* 2019. URL: `https://blockonomi.com/sharding/`.

[42] LastPass. *The Password Exposé: 8 truths about the threats – and opportunities – of employee passwords.* Tech. rep. 2017, p. 17. URL: `https://lp-cdn.lastpass.com/lporcamedia/document-library/lastpass/pdf/en/LastPass-Enterprise-The-Password-Expose-Ebook-v2.pdf`.

[43] EthGasStation. *What’s the Maximum Ethereum Block Size?* 2019. URL: `https://ethgasstation.info/blog/ethereum-block-size/`.

# A Deployed Smart Contracts and Front-End Application

We deployed an example of the proposed solution to the Ethereum blockchain and IPFS network, and anyone can interact with. The following list provides links to useful resources to interact with the system.

**Ethereum Smart Contracts:**

- Block Explorer to inspect the Registration Authority smart contract
  - `https://rinkeby.etherscan.io/address/0xFe4052Afe151a3656eE10579f80a8cA60E47a80D`
- Block Explorer to inspect the Election Factory smart contract
  - `https://rinkeby.etherscan.io/address/0xe0532CC114393aA7DFdc2a339BC927C279B80D25`

The front-end application has a reference to both the smart contracts above in the source code, and is configured to interact with them. The application is deployed to the IPFS network with the hash `QmdKXe8ZzFZeYW1PX4EatQgQD8RaLbGuPdRRnuH8nQik31`.

The application files can be retrieved using a dedicated IPFS client, or a browser using a gateway. Gateways allows users to retrieve IPFS content via HTTP connections directly in the browser. This is the preferred way of using the front-end applications.

As an example, the application can be opened using a gateway with the following link: `https://gateway.pinata.cloud/ipfs/QmdKXe8ZzFZeYW1PX4EatQgQD8RaLbGuPdRRnuH8nQik31`.

A list of public IPFS gateways can be found at `https://ipfs.github.io/public-gateway-checker/`, and all of them can be used to access the application using the hash above.

To inspect a specific Election smart contract with a block explorer, the Election address can be retrieved from the page URL when visiting the election page with the front-end application. The address can then be searched for at `https://rinkeby.etherscan.io/`.