ethVote: Towards secure voting with distributed ledgers

Johannes Mols
Aalborg University
Copenhagen, Denmark
jmols17@student.aau.dk

Emmanouil Vasilomanolakis
Aalborg University
Cyber Security Network
Copenhagen, Denmark
emv@es.aau.dk

***Abstract*—The topic of performing safe and secure elections is a long-standing debate. Regardless, of the various attempts for electronic or Internet-based voting, the majority of countries still use paper ballots. Nevertheless, with major advancements occurring over the last years in both cryptography and distributed ledgers we believe that there is space now for re-investigating this area. In this paper, we propose *ethVote* an Internet voting system that makes use of the Ethereum blockchain, state of the art cryptographic mechanisms and a P2P-based front-end to ensure a secure voting process. In addition, we provide an open-source proof of concept implementation that features the majority of the needed components for securely using *ethVote*. Our proposal is tested both in terms of unit testing, requirement verification, and with regard to the feasibility to perform such an operation in a public distributed ledger.**

I. 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 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 relatively secure over time, as every conceivable method of fraud has likely been tried and detected, and since defended against. Nevertheless, tampering with paper-based elections is possible, but requires tremendous effort, and many helpers are needed without anyone exposing the fraud. As a result, most countries nowadays are still using paper-based ballots.

However, a few countries have started experimenting with electronic voting despite the potential risks. Especially with Internet voting, instead of electronic voting machines, there are many advantages: it is cheaper, faster and more efficient at counting votes, and could bring higher voter turnout [1]. For example, Estonia became the first nation in the world to hold legally binding general elections over the Internet in 2005. Even though no attacks on the Estonian system have been detected as of today, many security experts have condemned the system. For instance, a group of researchers was able to manipulate votes and vote totals by installing malware on both personal computers and the voting servers themselves [2].

This paper 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 to traditional paper-based elections. For this, we take into account the explosion of research in the area of distributed ledgers [3] and cryptography for privacy preserving operations [4].

The core contribution of this paper is *ethVote*, a solution for Internet voting, which takes into consideration the advances in cryptography as well as the blockchain technology, and builds on top of them. In contrast to existing work, we carefully build a system that takes into account all the needed requirements (cf. Section III) for voting and establishes trust in the different stages of the voting process. We present the architecture and the technologies of our system and continue with a PoC that is open source and available to use [5]. Moreover, we discuss the current shortcomings and the future work for *ethVote*.

The remaining of this paper is structured as follows. In Section II we present the related work and in Section III we examine the requirements for a secure voting digital solution. Furthermore, Section IV presents our proposal ethVote; a blockchain-based Internet voting system. Subsequently, Section V discusses our PoC, while Section VI shows the evaluation of the proof of concept. Lastly, Section VII concludes this paper and presents our future work.

II. Related Work

Elections are worthless without the consensus among voters that their votes are cast securely and anonymously, and that the final count accurately represents their will. 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.

A. Paper Ballots and Electronic Voting Machines

Paper-based voting systems are the most secure way of conducting an election. This is due to two aspects [6]. First, voters know that the ballot accurately represents their intent because they can examine the ballot themselves. Second, voters know that there is a physical record of their vote,

[1]https://www.britannica.com/topic/election-political-science
[2]https://www.eiu.com/topic/democracy-index

which is difficult to destroy or change without leaving physical evidence.

However, there are significant disadvantages to paper systems. One of them is the cost of printing and distributing the ballots, organization of the election process, and the compensation of election helpers. Furthermore, it is inefficient and takes time to manually count every single ballot. Audits are only possible by manually re-counting a statistically significant sample by hand. Moreover, not every country has the capability to employ enough election helpers, or have the money to pay them. Instead, countries with high populations such as the United States, Brazil and India make heavy use of Electronic Voting Machines, which are discussed below.

Electronic Voting Machines are mechanical or electronic machines that take care of casting and counting votes. 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, both types of machines have been proven to be insecure by multiple independent works [7] [8] [9]. Due to the consensus among researchers about the security risks that come with EVM's, we consider them out of the scope of this paper.

B. Internet 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). We review different types of i-Voting systems that are designed to be used for actual elections.

1) Estonian i-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. The country is one of the most advanced digital societies in the world, with 99% of government services being available online, and approximately 44% of voters voting from home[3].

The Estonian legislation specifies that voters authenticate themselves with government-issued ID cards, which contain electronic data used for authentication and legally binding signatures. In addition, the online elections are held 4 to 6 days before Election Day, giving voters the chance to vote again on Election day and invalidate their i-Vote. Voters are also allowed to change their vote during the online election period to prevent vote selling or voting under coercion. The principle is that a traditional vote always has the priority over an online vote. Having the online election prior to Election day makes it possible to invalidate all online votes in case there are reasons to believe the system was compromised [10].

A group of international experts has observed Estonia's municipal elections in October 2013 and released a detailed report with their findings [2]. The group identified serious weaknesses both in the architecture of the system, and the procedures. Furthermore, they created a mock election setup using the published source code, and were able to develop both client-side and server-side malware that could change votes freely. The researchers concluded that Estonia should withdraw the online voting system immediately.

2) BroncoVote: BroncoVote [11] is a proof-of-concept that uses smart contracts on the Ethereum blockchain to securely store encrypted votes, and tally them using a homomorphic encryption scheme. The usage of blockchain allows to eliminate the vulnerabilities that are introduced by regular servers.

However, there are several flaws. First, there is a usage of an external, unsecured encryption server to circumvent the maximum size of 256 bits for an integer in Ethereum smart contracts. This is problematic because 256-bit integers cannot store sufficiently large asymmetric encryption keys (needed for homomorphic encryption schemes). This was proven by a researcher that was able to compute private keys, using the public keys, that were used in the Moscow City Duma election in 2019 [12]. Furthermore, BroncoVote has an insufficiently secure registration process and a flawed authentication of voters. Lastly, the system is not End-to-End verifiable due to the use of an external encryption server.

3) Hääl: Hääl [13] is another Ethereum-based voting system. In addition to homomorphic encryption, it uses zero-knowledge proofs to verify the correctness of votes without revealing the information contained in them. While this work is more sophisticated than BroncoVote, it has a flawed registration method[4].

4) Polys: Polys is a start-up company launched by Kaspersky, which is building a secure online voting system with the help of the Ethereum blockchain. To the best of our knowledge, it is the only blockchain-based project that is in a fully-functional state. Besides Ethereum, Polys is using Shamir's Secret Sharing scheme to distribute keys and ElGamal's homomorphic encryption scheme to encrypt votes. While the system appears to work as expected, the corresponding whitepaper [14] does not go into much detail about the specific implementation, and no source code has been released as of the time of writing. It is impossible to verify Polys' claims about security if no third-party is allowed to review the source code. Furthermore, the company uses a private Ethereum blockchain, where representatives serve as miners [15]. This might open up possibilities to attack the network if it consists of only a small number of nodes.

III. Requirements

Wang et al. proposed a list of requirements for i-Voting systems [16], differentiating between core and additional requirements. For the scope of our paper, we utilize both sets of requirements [16], along with our own addition (RQ16) as seen below. These will be the basis for our proposal *ethVote*.

- *RQ1: Correctness*. All votes are correctly counted. All valid votes need to be counted and any unauthorized or unauthenticated votes cannot be counted.
- *RQ2: Privacy*. No one except the voter themselves can know the voter's ballot choices.

[3]https://e-estonia.com/solutions/e-governance/i-voting/

[4]https://github.com/eddieoz/haal/issues/5

- *RQ3: Unreusability*. A voter cannot cast a vote twice.
- *RQ4: Eligibility*. Only authorized voters can vote.
- *RQ5: Robustness*. System functions even with a number of misbehaved voters (or partial failure of the system).
- *RQ6: Verifiable*. A voter can verify that their ballot has been counted.
- *RQ7: Usability*. A broad term that plays an important role in the success of the system.
- *RQ8: Fairness*. No partial results are computed before the end of the election.
- *RQ9: Uncoercability*. Voters cannot be forced to cast their vote in an unintended way. Votes cannot be sold.
- *RQ10: Efficiency*. The complexity of computation given the vast amount of voters.
- *RQ11: Mobility*. Votes can be cast using mobile devices.
- *RQ12: Vote-and-Go*. Voters can go offline after submitting their ballot. No further computation from the voters' side is required.
- *RQ13: Universal Verifiability*. Anyone can verify the final vote count.
- *RQ14: E2E-Verifiable*. End-to-End Verifiable systems produce a receipt to the voter that convinces them of the correctness of their vote without revealing the choice of the vote on the receipt.
- *RQ15: Practicality*. The system does not assume the security of any of the components.
- *RQ16: Asymmetric key pairs for homomorphic encryption*. Recommended minimum key length for asymmetric keys should be used (i.e. 2048-bits [17]).

IV. ethVote

We propose *ethVote*, an Internet Voting system that is based on Ethereum smart contracts. *ethVote* uses the Paillier homomorphic encryption scheme to encrypt and tally votes, as well as non-interactive zero-knowledge proofs to verify the correctness of encryptions and decryptions of votes and tallies. The system consists of mainly two entities: the smart contracts on the Ethereum blockchain and a front-end web application, as depicted in Figure 1.

A. Smart Contracts

The Ethereum blockchain is an important part of the system. Three separate smart contracts control the logic of the entire voting system. They store a list of eligible voters, organizational details of every election, and the encrypted votes of each user per election. The majority of requirements can be fulfilled by designing smart contracts in a secure way.

1) Registration Authority: This smart contract stores the Ethereum address of every voter that is allowed to participate in the elections. Each address can have personally identifiable information stored alongside the address, which should be encrypted in an actual election system. The functions to register or unregister an address can only be accessed by the address of the Registration Authority. This organization is solely responsible for maintaining a list of eligible voters, and has no control over any of the elections. We decided to require voters to authenticate themselves to the Registration Authority by physically showing up and presenting a valid passport or ID card, as online authentication might introduce vulnerabilities to the system.

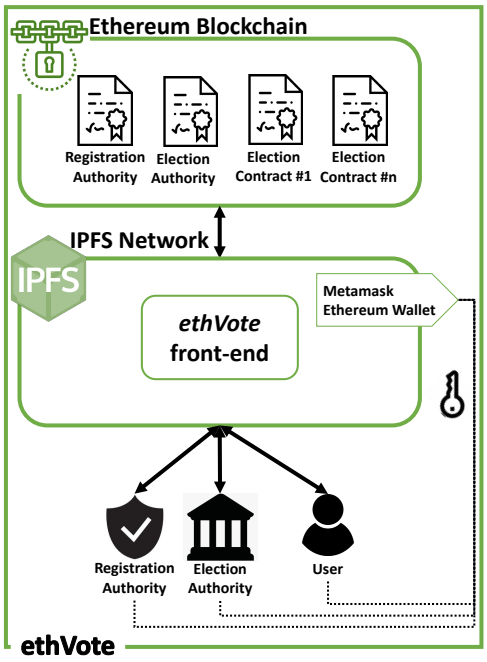

Fig. 1. *ethVote* System Architecture

2) Election Factory: This smart contract is responsible for spawning new Election contracts, and maintaining a list of their Ethereum addresses. This is necessary so that every voter can reliably determine where to find which election. The function to create new elections can only be accessed by the Election Authority, which may or may not be the same as the Registration Authority. When a new election is created, it is given a name and description, a start/end-time in seconds since the Epoch, and the public encryption key for the homomorphic encryption.

3) Election: This smart contract contains the core logic of the system. Three stages define the available functionality: *before*, *during*, and *after* the election. Function modifiers that check the current time against the predefined start/end time, control whether functions are allowed to be called, and by whom.

a) Before the election: the Election Authority can add candidates to the election. Voters can see which candidates are available before the election starts.

b) During the election: the list of candidates is fixed and voters can submit their encrypted votes using the public key stored in the smart contract. Each vote consists of a list of encrypted 0's or 1's, indicating whether each candidate received a vote from the voter or not. Only one "1" is allowed per vote. When tallying, the encrypted numbers are tallied and the decryption of that number is the total votes for that candidate. Before a vote is accepted into the smart contract, the Registration Authority is queried with the voter's address to verify that they are allowed to cast a vote. The data structure

to store votes is a mapping between Ethereum address and vote. When a voter changes their vote, the existing vote is overwritten.

c) After the election: anyone can retrieve a list of addresses that submitted a vote, and then their encrypted vote. This allows anyone to compute the final encrypted tally for each candidate. Using zero-knowledge proofs, the Election Authority can then prove that the decryption of each tally is accurate, without revealing the private key.

B. Homomorphic Encryption

ethVote utilizes homomorphic encryption to be able to perform operations over encrypted data. A cryptosystem is homomorphic if it has the property that the multiplication of two ciphertexts is the encrypted sum of the plaintexts associated with the ciphertexts. That is:

$$\varepsilon(m) \otimes \varepsilon(m') = \varepsilon(m \oplus m') \tag{1}$$

ε is an encryption algorithm, and $\otimes$ and $\oplus$ are operations on the ciphertexts and plaintexts, respectively [18].

The advantage of this property is that the computation can be performed on encrypted data, for which only the public keys need to be known. Ron Rivest, argues that a homomorphic voting system could be the future, as it is highly transparent and auditable [19]. 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 tally results in a certain decrypted tally. To do this, zero-knowledge proofs can be used.

Hence, *ethVote* uses homomorphic encryption to encrypt votes and tally the results. Since we are using a blockchain, each vote is also automatically publicly available, while still protecting the voters privacy. When a new election is created, a key-pair with private and public key is automatically generated by the front-end application. 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 using the Paillier cryptosystem. 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.

C. Zero-knowledge Proofs

Homomorphic encryption plays a central role in the architecture of *ethVote*, as it greatly improves voter privacy. However, 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 cannot encrypt votes themselves. First, this is because it is computationally expensive, and blockchains are not made to perform complex calculations. Second, sending unencrypted votes to a smart contract would expose the vote to the public, as the blockchain is fully transparent.

That said, and with *ethVote* being an open source project, it would be trivial for an attacker to determine the encryption methods used and build software 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", or vote for multiple candidates. Since the result is encrypted (i.e., 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 [20]. 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 *ethVote* because it allows voters to convince the smart contract, and therefore everyone else, that their encrypted vote follows the rules and is valid (without revealing the actual vote). Furthermore, it can be used to prove that the published result of an election is the actual decryption of the encrypted tally.

Non-interactive zero-knowledge proofs do not 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 encrypt their votes correctly, and the Election Authority to decrypt the results correctly.

D. Front-end application

The front-end application is hosted on the *InterPlanetary File System* (IPFS)[5] for additional robustness. A regular web server could easily be attacked and taken down with a DDoS attack, while a peer-to-peer network such as IPFS can guarantee the availability of application files.

V. Proof of Concept (PoC)

We implemented a proof-of-concept for *ethVote*. The result is a fully working election system running on the Ethereum test network, and a front-end application that is hosted on the *InterPlanetary File System* (IPFS) network, which allows users to interact with the smart contracts on the blockchain. The source code of both the smart contracts (implemented in *Solidity*) and the front-end application is publicly available on GitHub [5] [21]. Unfortunately, the current version of the PoC does not implement zero-knowledge proofs (see Section VII). Nevertheless, we do provide a pseudo-code implementation for this (see Section V-B).

[5]https://ipfs.io

A. ethVote Smart Contracts

The general structure of the three smart contracts is presented with Smart Contract 1, 2, and 3 respectively. Only the most important parts are shown, and the full contracts can be found on GitHub. The purpose of each smart contract was outlined in Section IV-A.

Smart Contract 1: Registration Authority

address public manager;
mapping(address → Voter) private voters;
address[] private votersReferenceList;
Struct *Voter*:
  bool isVoter;
  address ethAddress;
  string personalInformation;
Modifier *restricted*:
  require(msg.sender == manager);
Constructor *constructor()*:
  manager := msg.sender;
Function *restricted registerOrUpdateVoter(address voter, string personalInformation)*:
  add or update voter details using their address *voter*;
Function *restricted unregisterVoter(address voter)*:
  remove voter address from *voters*;

1) Registration Authority: The Ethereum address of the contract manager (Registration Authority) is publicly stored in the variable *manager*. The constructor is executed when the smart contract is created and sets the creator of the contract (*msg.sender*) as the manager.

The function modifier *restricted* defines that the caller of a function with this attribute has to be the contract manager. This prevents anyone else from executing the function.

The functions *registerOrUpdateVoter* and *unregisterVoter* both have this function modifier. The former receives a voter's address and optional personal information in the parameter, and adds the address to the list of registered voters. The latter removes a specified address from that register.

The data structure of this register consists of two variables: a *mapping* between addresses and Voter structs (objects), and an array of addresses. The reason for this is that Solidity does not store the keys associated with a value in a mapping[6]. This means that the key has to be known to retrieve the value. While a mapping alone is sufficient to look up a voter registration, we cannot retrieve a list from it. In order to store encrypted votes in Smart Contract 3, this is necessary.

2) Election Factory: Similarly to the first smart contract, the manager of the contract is defined in the constructor and stored in *factoryManager*. A reference to the first smart contract's address is also provided, which is passed to all Election contracts, which can then use the Registration Authority to verify voter registrations. The function *createElection* is only accessible to the contract manager, and creates a new Election contract with specific details such as the title, time limits, and public encryption key. The address of the newly created contract is stored in *deployedElections*. All deployed elections can be found using the function *getDeployedElections*.

Smart Contract 2: Election Factory

address public factoryManager;
address public registrationAuthority;
address[] public deployedElections;
Modifier *restricted*:
  require(msg.sender == factoryManager);
Constructor *constructor(address ra)*:
  factoryManager := msg.sender;
  registrationAuthority := ra;
Function *restricted createElection(details)*:
  deploy election contract with *details* and add address to *deployedElections*;
Function *getDeployedElections()*:
  return deployedElections;

3) Election: The details about the election are stored in the public variables *title*, *description*, *startTime*, *timeLimit*, and *encryptionKey*. Furthermore, the addresses of the Election Factory (EF) and Registration Authority (RA), and the address of the Election Authority (EA) are stored. The EA is considered the administration of the Election and can access restricted functions.

Three more function modifiers regulate at what time which functions are accessible. For example, it cannot be allowed to cast a vote before or after an elections. The current time in Ethereum smart contracts is the last block time, so it can vary from real time by a few seconds[7]. This is acceptable in this PoC, as we expect elections to last several hours or days. The most important function is to *vote*. It is only accessible during an election, and expects an encrypted vote in string format. First, the RA contract is checked whether the sender's address is a registered voter. If so, the vote is added to the *votes* mapping, using the storage pattern that we described in Section V-A1.

B. Zero-knowledge Proofs

An implementation of zk-SNARKs (Zero-knowledge Succinct Non-Interactive Argument of Knowledge) has to satisfy three requirements:

- *Completeness*: if the statement is true then a prover can convince a verifier
- *Soundness*: a dishonest prover cannot 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 need to be constructed. A prover can then run their secret parameters

[6]https://ethereum.stackexchange.com/a/15341/55206

[7]https://etherscan.io/chart/blocktime

Smart Contract 3: Election

```
address public eleFactory, eleManager, regAuth;
string public title, description;
uint public startTime, timeLimit; // Unix Time
Option[] public options;
string public encryptionKey;
uint[] public publishedResult;
mapping(address → Vote) private votes;
address[] private votesReferenceList;
Struct Option:
    string title;
    description description;
Struct Vote:
    uint listPointer;
    string encryptedVote;
Modifier manager:
    require(msg.sender == eleManager);
Modifier beforeElection:
    require(now < startTime);
Modifier duringElection:
    require(now > startTime && now < timeLimit);
Modifier afterElection:
    require(now > timeLimit);
Function duringElection vote(string vote):
    if isRegisteredVoter(msg.sender) == true then
        votes[msg.sender].encryptedVote = vote;
        if hasVoted(msg.sender) == false then
            add voter address to votesReferenceList;
        return true;
Function hasVoted(address voter):
    return whether votesReferenceList contains a vote
     from the specified address;
Function isRegisteredVoter(address voter):
    ra := RegistrationAuthority(regAuth);
    return ra.isVoter(voter);
```

through the circuit and retrieve a proof that can be presented to the verifier. The proof does not contain the secret parameters. The logic of such circuits are generally simple and only verify that two values are equal, with a limited set of arithmetic operations available. The resulting arithmetic circuits, however, are highly complex and almost impossible to construct by hand.

A popular tool for this is *ZoKrates* [22]. It provides a high-level programming language which can be compiled to arithmetic circuits. It is also capable of generating Ethereum smart contracts, which can be used to verify proofs directly on the blockchain. The proofs can be calculated with the tool itself, or a JavaScript language binding [23]. This allows web applications to generate proofs and submit them to the verifying smart contract.

Unfortunately, the ZoKrates language is 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. Furthermore, there is only a limited number of standard functions available, requiring developers to write low-level code. When dealing with special encryption schemes and large numbers, both of these facts become a major hurdle. Because of this, we were not able to implement the required zero-knowledge proofs for the PoC. However, we believe it is possible if the tools to create them become more powerful. In the following, we provide the pseudo-code that would satisfy the requirements.

To prove the correctness of decryption of the final tally, the zero-knowledge proof, depicted in Algorithm 1, would suffice.

Algorithm 1: Zero-knowledge proof of a tally

```
Input: public decryptedResult (dR), public
       encryptedResult (eR), private privateKey (pK)
Function main(dR, eR, pK):
    calculatedResult := decrypt(eR, pK)
    if dR == calculatedResult then
        return true
    else
        return false
Function decrypt(eR, pK):
    return L(eR^λ mod pK.n^2) * pK.μ mod pK.n
Function L(x, pK):
    return (x − 1) / pK.n
```

This proof requires the Election Authority to decrypt the final tally on locally, 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. By implementing a Paillier decryption method in ZoKrates, and having sufficient variable sizes, this is possible. When the authority 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. Since the voter only has the public key for encryption, the voter cannot prove a 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 could not simply reproduce the encryption in the proof, as the private key is unavailable to a voter. This can be solved by storing unproven votes in a separate list, where the Election Authority would verify each vote using the private key. If a vote is proven to be valid, it can be added to the list of proven votes. This process is illustrated in Algorithm 2. Note that Algorithm 2, assumes the presence of the functions $decrypt$ and L from Algorithm 1, but was excluded here due to space constrains.

C. ethVote Front-end

The smart contracts, homomorphic encryption, and zero-knowledge proofs provide all the critical security measure-

Algorithm 2: Zero-knowledge proof of a vote

Input: *public* encryptedVote (eV), *private* privateKey (pK)
Function *main(eV, pK)*:
  sum := 0
  for ($i := 0;\ i < len(eV);\ i := i + 1$)
    sum := sum + decrypt(eV[i])
  if *sum* == 1 **then**
    return true
  else
    return false

ments to satisfy the requirements for a secure i-Voting system, except *Usability* (RQ 7) and *Mobility* (RQ 11). These requirements can be satisfied with the front-end application. Furthermore, the front-end application has to provide a secure way of authenticating voters.

Authentication is provided by the *MetaMask* browser extension [24]. 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.

For the implementation, we used the open-source library *React* to create the user interface, *Semantic UI React* to style the interface, a JavaScript implementation of the Paillier cryptosystem called *paillier-js*, and the Ethereum JavaScript API called *web3.js*, which is used to connect to the Ethereum network without running a full node.

VI. Evaluation

To evaluate the PoC, we created unit tests of all smart contracts, specify how each requirement is met, and evaluate the running cost on the Ethereum network.

A. Unit testing and security analysis

To conduct unit tests of smart contracts, they have to be tested offline, as the actual Ethereum networks influence the contract's functionality. We do this with *Ganache*[8], a local Ethereum blockchain specifically created for development.

The testing script is written in JavaScript, using the *mocha* test framework, and the library *web3.js*. Note that front-end application is using the same technologies, which results in almost identical code for interaction with the smart contracts.

Every functionality of all three smart contracts is tested in one or more test cases, which allows us to immediately notice any breaking changes during development. We scanned the smart contracts using *MythX*[9], which yielded 5 warnings about implicit loops over unbounded data structures. This could cause a DoS if too much data is passed into a function. However, this is unlikely to happen due to the gas limits.

[8]https://www.trufflesuite.com/ganache
[9]https://mythx.io/

B. Requirement verification

Table I summarizes how each requirement from Section III is satisfied. We assume a working implementation of zero-knowledge proofs for this.

Requirement	Blockchain	Smart Contracts	Hom. Encryption	Zero-k proofs	Front-end app
1. Correctness	●	●	○	●	○
2. Privacy	○	○	●	○	○
3. Unreusability	○	●	○	○	○
4. Eligibility	○	●	○	○	○
5. Robustness	●	●	○	●	●
6. Verifiable	●	●	○	○	●
7. Usability	○	○	○	○	●
8. Fairness	○	●	○	○	○
9. Uncoercability	○	●	○	○	○
10. Efficiency	○	○	○	○	●
11. Mobility	○	○	○	○	●
12. Vote-and-Go	●	●	○	○	○
13. Universal Verifiability	●	●	●	●	○
14. E2E-Verifiable	○	○	○	●	○
15. Practicality	●	○	○	○	●
16. 2048-bit keys	○	●	○	○	●

TABLE I
Requirement Verification: ●= satisfies requirement, ○= does not satisfy requirement

The Ethereum blockchain serves as a secure storage of all election-related data. While it cannot satisfy any requirement by itself, it is not possible to fulfill *Correctness*, *Robustness*, *Verifiability*, *Vote-and-Go*, *Universal Verifiability*, and *Practicality* (RQ 1, 5, 6, 12, 13, 15) without it. The smart contracts build on top of the Blockchain, and provide the core logic of the voting system. The PoC implementation solely fulfills *Unreusability*, *Eligibility*, *Fairness*, and *Uncoercability* (RQ 3, 4, 8, 9). Since it is closely connected to the blockchain, all requirements mentioned above are also fulfilled by the smart contracts, except for *Practicality* (RQ 15), which is only fulfilled by the smart contracts. Furthermore, the smart contracts can store votes encrypted with *2048-bit keys* (RQ 16). However, the size of those encryptions can lead to high gas costs (see Section VI-C).

Homomorphic encryption plays an important role in the *Privacy* (RQ 2) of voters, and allows the system to be *Universally Verifiable* (RQ 13). Zero-knowledge proofs can convince the public that all cryptographic operations are correct. They therefore contribute to the *Correctness*, *Robustness*, and *Universal Verifiability* (RQ 1, 5, 13) of the system. The system is also *End-to-End Verifiable* (RQ 14) due to this technology.

Lastly, the front-end application offers users an interface to the smart contracts. It therefore contributes to the *Robustness*, *Verifiability*, and *Practicality* (RQ 5, 6, 15) of the system. It is also solely responsible for the *Usability*, *Efficiency*, and *Mobility* (RQ 7, 10, 11). Furthermore, it assists the usage of *2048-bit keys* (RQ 16) with the encryption and decryption on the client-side.

C. Gas costs

Table II presents gas cost measurements of all actions available in the smart contracts. All tests were conducted on the Rinkeby test network. The analysis shows that elections with a large number of candidates can get expensive quickly. This is because a vote consists of an individually encrypted vote for each candidate in an election, meaning that the size of encrypted votes will grow as the candidate list grows.

Action	Cost in ETH	Cost in USD
Registration Authority		
Contract creation	0.001154	0.26
Register voter without data	0.000101	0.02
Register voter with data	0.000177	0.04
Election Factory		
Contract creation	0.002431	0.55
Election creation with 2048-bit key	0.005048	1.15
Election		
Add option / candidate	0.000132	0.03
Vote (1 candidate in election)	0.000132	0.03
Vote (2 candidates in election)	0.001347	0.30
Vote (5 candidates in election)	0.002551	0.58
Vote (10 candidates in election)	0.008058	1.83

TABLE II
GAS COSTS OF PROOF-OF-CONCEPT

Germany has spent 92 million EUR on the last federal election in 2017[10]. The number of parties on the ballot vary by state, but on average, there were 19. Using the values in Table II and a linear regression model, we can estimate the cost for one vote to be 0.0154 ETH. There were approximately 47 million votes cast, resulting in a total estimated cost of 723,800 ETH just for votes. As of March 2020, this equals to 152.3 million EUR. This amount is higher than the aforementioned German elections cost. However, this can be significantly reduced by: i) using a private blockchain, ii) the updated Ethereum 2.0. version, iii) reducing the private key sizes; we do acknowledge, however, that i) and iii) come with respective disadvantages. Lastly, the current numbers do not take into account the environmental impact of paper ballots.

VII. CONCLUSION

In this paper, we proposed *ethVote* a system for i-voting that makes use of distributed ledgers, the Paillier homomorphic encryption scheme, zero-knowledge proofs and a P2P network (i.e. IPFS) to cope with the various challenges and attacks that come hand in hand with the voting process. There is still a long road ahead towards a system that can be deployed for real world elections. Nevertheless, we are confident that *ethVote* is a step towards this goal.

Future work includes the addition of the zero-knowledge proof implementation to the PoC. Moreover, we plan to further investigate the current limitations on the number of candidates and the high gas costs, which comes as a result of the encryption key sizes and the Ethereum's block size limitations.

[10]https://www.saarbruecker-zeitung.de/nachrichten/politik/inland/bundestagswahl-kostet-so-viel-wie-noch-nie_aid-2540033

REFERENCES

[1] N. Kersting and H. Baldersheim, *Electronic Voting and Democracy: A Comparative Analysis*. Palgrave Macmillan Ltd, 2004.

[2] D. Springall, T. Finkenauer, Z. Durumeric, J. Kitcat, H. Hursti, M. MacAlpine, and J. A. Halderman, "Security Analysis of the Estonian Internet Voting System," in *Proceedings of the ACM Conference on Computer and Communications Security*, 2014. [Online]. Available: https://estoniaevoting.org/

[3] X. Li, P. Jiang, T. Chen, X. Luo, and Q. Wen, "A survey on the security of blockchain systems," *Future Generation Computer Systems*, 2017.

[4] A. Acar, H. Aksu, A. S. Uluagac, and M. Conti, "A survey on homomorphic encryption schemes: Theory and implementation," *ACM Computing Surveys (CSUR)*, vol. 51, no. 4, pp. 1–35, 2018.

[5] J. Mols, "ethVote," 2019. [Online]. Available: https://github.com/johannesmols/ethVote

[6] D. Evans and N. Paul, "Election Security: Perception and Reality," *IEEE*, p. 8, 2004. [Online]. Available: https://ieeexplore.ieee.org/document/1264850/

[7] S. Wolchok, E. Wustrow, J. A. Halderman, H. K. Prasad, A. Kankipati, S. K. Sakhamuri, V. Yagati, and R. Gonggrijp, "Security analysis of India's electronic voting machines," in *Proceedings of the ACM Conference on Computer and Communications Security*, 2010.

[8] M. Blaze, H. Hursti, M. Macalpine, M. Hanley, J. Moss, R. Wehr, K. Spencer, and C. Ferris, "DEFCON 27 Voting Machine Hacking Village," 2019. [Online]. Available: https://media.defcon.org/DEFCON27/voting-village-report-defcon27.pdf

[9] "Voting System Security and Reliability Risks," Brennan Center for Justice, New York, Tech. Rep., 2016. [Online]. Available: https://www.brennancenter.org/sites/default/files/analysis/Fact_Sheet_Voting_System_Security.pdf

[10] E. Maaten, "Towards remote e-voting: Estonian case," *Conference: Electronic Voting in Europe - Technology, Law, Politics and Society*, 2004.

[11] G. G. Dagher, P. B. Marella, M. Milojkovic, and J. Mohler, "BroncoVote: Secure Voting System using Ethereum's Blockchain," *Proceedings of the 4th International Conference on Information Systems Security and Privacy (ICISSP)*, 2018.

[12] P. Gaudry, "Breaking the encryption scheme of the Moscow internet voting system," Tech. Rep., 2019. [Online]. Available: https://arxiv.org/abs/1908.05127

[13] E. Junior, Osorio, "Anonymous Electronic Voting System on Public Blockchains," 2018. [Online]. Available: https://github.com/eddieoz/haal

[14] Polys, "Polys - Online Voting System." [Online]. Available: https://polys.me/assets/docs/Polys_whitepaper.pdf

[15] ——, "What's the difference between Polys and Google Forms?" [Online]. Available: https://docs.polys.me/en/articles/1641300-what-s-the-difference-between-polys-and-google-forms

[16] K.-H. Wang, S. K. Mondal, K. Chan, and X. Xie, "A Review of Contemporary E-voting: Requirements, Technology, Systems and Usability," *Ubiquitous International*, 2017.

[17] B. Kaliski, "Twirl and RSA Key Size," 2003. [Online]. Available: https://web.archive.org/web/20170417095741/https://www.emc.com/emc-plus/rsa-labs/historical/twirl-and-rsa-key-size.htm

[18] N. Pettersen, "Applications of Paillier's Cryptosystem," Master thesis, Norwegian University of Science and Technology, 2016. [Online]. Available: https://pdfs.semanticscholar.org/b3c9/5d839a48418e3674380a76b8fe6c960d37c7.pdf

[19] R. Rivest, "Was YOUR vote counted? (feat. homomorphic encryption)," 2016. [Online]. Available: https://youtu.be/BYRTvoZ3Rho

[20] M. Blum, P. Feldman, and S. Micali, "Non-interactive zero-knowledge and its applications," in *Providing Sound Foundations for Cryptography: On the Work of Shafi Goldwasser and Silvio Micali*, 2019, pp. 329–349.

[21] J. Mols, "ethVote-react," 2019. [Online]. Available: https://github.com/johannesmols/ethVote-react

[22] ZoKrates, "Zokrates: A toolbox for zksnarks on ethereum," 2019. [Online]. Available: https://github.com/Zokrates/ZoKrates

[23] ——, "Zokrates-js: Javascript bindings for zokrates project." 2019. [Online]. Available: https://github.com/Zokrates/zokrates-js

[24] W.-M. Lee, "Using the metamask chrome extension," in *Beginning Ethereum Smart Contracts Programming*. Springer, 2019, pp. 93–126.