frontiers | Frontiers in Blockchain

TYPE Technology and Code
PUBLISHED 01 March 2023
DOI 10.3389/fbloc.2023.1105119

OPEN ACCESS

EDITED BY
Francesco Tiezzi,
Università degli Studi di Firenze, Italy

REVIEWED BY
Ivan Mercanti,
University of Perugia, Italy
Andrea Morichetta,
University of Camerino, Italy

*CORRESPONDENCE
Damián López,
dlopez@dsic.upv.es

SPECIALTY SECTION
This article was submitted to Smart Contracts,
a section of the journal
Frontiers in Blockchain

RECEIVED 22 November 2022
ACCEPTED 16 February 2023
PUBLISHED 01 March 2023

CITATION
Larriba AM and López D (2023), A Solidity implementation of TAVS.
Front. Blockchain 6:1105119.
doi: 10.3389/fbloc.2023.1105119

A Solidity implementation of TAVS

Antonio M. Larriba and Damián López*

vrAIn-Valencian Research Institute for Artificial Intelligence, Universitat Politècnica de València, Valencia, Spain

We present a Solidity smart contract implementation of the TAVS e-voting protocol. The Two Authorities Electronic Voting Scheme (TAVS) is a voting scheme that achieves universal verifiability with a reduced time-complexity both for the elector and the voting system. TAVS security derives from the RSA cryptosystem it employs, and the assumption of two entities that do not share information. We present a Solidity implementation which replaces one of these entities with an immutable smart contract in Ethereum based networks. By doing so, our implementation extends the security properties of TAVS and achieves a higher degree of resilience, verifiability, and availability. We open source the code of the implementation.

KEYWORDS

electronic vote, secret sharing, blind signatures, Solidity, blockchain, smart contracts

1 Introduction

Electronic voting has been heavily studied given its crucial role in democratic societies. Electronic voting is generally based on cryptography to ensure both the privacy of the voter, as well as the legitimacy of the scheme. The use of different cryptographic primitives such as: blind signatures (Chaum, 1983), homomorphic cryptography (Moore et al., 2014), ring signatures (Rivest et al, 2006) or zero-knowledge proofs (Goldwasser et al., 1989) is intended to prevent double voting while ensuring elector's privacy and the integrity of the tally. We refer the interested reader to (Mursi et al., 2013) as a recommended survey on e-voting to learn more about these techniques.

In (Larriba et al., 2020), the authors present a Two Authorities Voting Scheme (TAVS), an e-voting protocol based on blind signatures and focused on efficiency. TAVS provides all the properties an e-voting scheme requires (verifiability, democracy, integrity, correctness, and privacy), while providing reduced time computational complexity when compared to other similar schemes. TAVS security is derived from the RSA cryptosystem used to implement blind signatures and from the fact that it assumes two unrelated entities that do not share information during the election. While there are many instances of an election where you can find two antagonistic entities that will never share information, there are also many scenarios where finding such unrelated entities might be impossible. Our implementation reduces the assumption of honesty, from two entities to one, without degrading the security properties of TAVS. The main motivation of our work is to reduce security assumptions. We replace the tallying authority by an immutable smart contract, solving this way the problem of finding two honest entities, since smart contracts are self-governed entities that only obey the source code. Our solution accomplishes these properties has been made public for auditability and to contribute to the open source community. This implementation is fully equivalent to TAVS except in the tallying property, since everything is public in blockchain, and by default anyone can see the votes before the election ends. We address this difference of the implementation with respect original proposal, by later presenting a solution to overcome this issue.

Since the Bitcoin whitepaper (Nakamoto, 2008) was published in 2008, blockchain technology adoption exploded. Blockchain has become one of the most relevant decentralized networks ever. Due to its decentralized consensus layer, Bitcoin was able to provide a global and secure payment network. From the user's perspective, the blockchain is seen as an immutable public ledger structured as a sequence of blocks. These blocks bundle transactions that contain information to update the ledger state. Hence, the blockchain can be seen as a state machine that is updated through atomic blocks.

This decentralized capability to update a global state such as the blockchain is a two-edged sword. To limit the implied risks, Bitcoin was designed to only support a small non-Turing complete script language. This fact made the Bitcoin network extremely stable, but also limited and only able to support simple value transfers. Ethereum (Ethereum, 2014) was introduced as a network with a Turing complete support language called Solidity. Solidity allows to implement custom and arbitrary functionality through smart contracts. Hence, Ethereum can be seen as a global network for distributed computation and not only as a payment network. To address the threats of malicious users collapsing the network, Ethereum introduced the concept of *gas*, forcing users to pay (in the same currency the network employs) for computing units.

The contributions of this work can be summarized as:

- We implement the TAVS voting scheme in Solidity without compromising any of its security properties.
- We demonstrate how by using blockchain technology and smart contracts, a higher degree of privacy can be achieved by removing trusted entities.
- We present a guide and a use case to prove the feasibility of our implementation.
- We open-source the code to allow further research on blockchain-based voting schemes.

In this work, we explore and demonstrate the smart contract capabilities to implement the TAVS voting scheme as a Solidity smart contract.

The rest of the paper is organized as follows. Section 2 covers the related works in the literature. Section 3 provides a short and concise summary of how TAVS operates and the votes are crafted. In Section 4 we present our Solidity implementation: the design decisions, the limitations in the Ethereum Virtual Machine (EVM) model, and the code organization. Next, we provide some working examples in Section 5 describing interaction with the implementation. Finally, we present our closing thoughts in Section 6.

2 Related work

Blockchain-based systems have become very popular as blockchain technology matured from a transactional system to a general and distributed consensus layer. Its positive effect on transparency and voter confidence issues (Moura and Gomes, 2017) also helped to bring blockchain into electronic voting. In this section, we review relevant theoretical works in the literature. We refer the interested reader to these surveys to learn more about the challenges of blockchain systems (Taş, 2020), and some of their implementations (Curran, 2018; Kshetri and Voas, 2018).

In (Hreiðarsson et al., 20182018), the authors introduce a smart contract based voting scheme. In this protocol, there are only two entities: electors, and election administration officers. Electors go trough an identification process that provides them with a unique wallet. Only official wallets are able to send a valid vote. Electors need to communicate through ballot smart contracts, which depend on the district, to cast their votes. These votes are verified by a Proof-of-Authority (PoA) network (also run by election administrators) external to the blockchain. If the verification succeeds, the PoA network adds the transaction containing the vote in the blockchain. The limitation of this approach is twofold: first, it requires a dedicated PoA network to scan the blockchain; and second, the election administrators have all the power in the system. Thus, the creation of the election, the privacy of users, and the validity of the votes depend on the administrators. Therefore, there is no real privacy of the electors nor distribution of the responsibilities.

The authors present in (Gioulis and Markantonakis, 20182018) a blockchain voting system similar to our approach. Their registration phase is also based on blind signatures, although the ballot structure is different, and requires an honest authority responsible for identifying the electors. The main difference with our implementation is that the verification and tallying of votes is carried out manually by the electors themselves. Since they do not employ smart contracts, they require to be involved in block production, and they have to operate in private blockchains as well. The implementation does not seem to be open sourced.

In (Yang et al., 2020), the authors propose a range voting protocol based on blockchain to structure the election process. All candidates are ranked (voted with tokens in the blockchain), and the candidate with a higher score wins. ElGamal (1985) and group-based encryption is employed to preserve elector's privacy. Thanks to the homomorphic properties of ElGamal, the final tally can be computed without decrypting individual votes. Hence, preserving the final elector's privacy. They do not use smart contracts, the blockchain is used as a secure public bulletin.

In (Gao et al., 2019), an e-voting protocol based on blockchain technology is presented. The protocol is based on a NP-complete problem (Niederreiter, 1985) and provides post-quantum resistance. The method also employs ring signatures to preserve the privacy of the sender. Votes are structured as transactions from electors to candidates. The protocol has an audit function that allows to detect fraudulent voters, and compute the tally, while respecting their privacy. This work does not consider smart contracts in the implementation of the protocol.

A blockchain-based election scheme was presented in (Chouhan and Arora, 2022). The authors present an implementation, built on the Hyperledger[1] blockchain framework, that it is compatible with most election setups and supports an unlimited number of electors. To keep the outcome of the elections private until the tallying phase, they employ Shamir's secret sharing scheme (Shamir, 1979). Votes are encoded as points of a polynomial which are distributed between a set of assumed honest authorities. When the voting phase ends, these authorities interpolate the resulting polynomial and recover the vote. To protect the elector's privacy, during the registration

1 https://www.hyperledger.org/

phase elector's identification is mapped to an anonymous identification. Unfortunately, this is only possible because Hyperledger is not completely transparent and does not produce decentralized networks, only permissioned distributed networks. The authors present a comprehensive explanation of the Hyperledger contracts, but the code is not open-sourced.

In (Onur and Arda, 2022), the authors present a smart-contract based election scheme for ranked voting. To the best of our knowledge, it is the most similar system to our implementation. Their protocol is also developed in Solidity for Ethereum compatible chains, and they do open-source their implementation. In this scheme, the privacy of the voter is obtained through Zero-Knowledge proofs (Goldwasser et al., 2019). During the registration phase, electors produce a commitment of their identity that is stored in a Merkle tree. Later on, during the voting phase, they can craft a zero-knowledge proof that shows they are eligible electors in the census, without revealing in which specific leaf of the Merkle tree their commitment is stored. To maintain the vote secret until the tallying phase, they employ a commit and reveal scheme. The main differences with our implementation are flexibility and scalability. First, the vote encoding of or implementation does not limit the election to ranked voting, multiple election systems should be supported. Secondly, and despite their great power, general zero-knowledge proof systems are computationally heavy. The number of potential electors is limited by the size of the Merkle tree. The size of the Merkle tree can of course be increased, but not without deeply affecting the computational resources needed to generate the proof.

To the best of our knowledge, we are one of the few works that implements and open-sources the code of a peer-reviewed election blockchain voting system based on smart contracts. Almost all the proposals in the literature are theoretical and do not leverage the computation layer of blockchain. In those protocols, blockchain is generally used simply as a messaging layer and/or a public bulletin board.

3 TAVS review

We devote this section to provide a summary of how TAVS is designed and how the ballot is constructed. Please note, that, unlike other voting systems, the corruption of one of the involved parties does not compromise the integrity of the voting scheme, but it could compromise the privacy of the electors. For this reason, we substitute the authority in charge of verifying and tallying votes with a smart contract. We refer the reader to (Larriba et al., 2020) for more details and security proofs on TAVS. For the rest of the article, we use the feminine she/her to refer to the elector, to make a clear distinction between the final user, and the parties involved in the election itself. This is done with the sole purpose of clarity.

TAVS is an electronic voting scheme that reduces the number of involved authorities to only 2 of them: an Identification Authority (IA) in charge of ensuring membership of the elector in the census; and, a Remote Polling Station (RPS) in charge of receiving, verifying and tallying the votes. In order to ensure the privacy of the elector, as well as the validity of the votes, TAVS employs a blind signature scheme. The blind signature scheme is built by taking advantage from the homomorphic properties of the RSA signature protocol (Rivest et al, 1983).

TAVS consists on three sequential steps. First, the elector carries out a pre-ballot generation step without interacting with any party. Then, she initiates an identification process with the IA by using blind signatures to obtain a valid ballot. Finally, she anonymously sends the ballot to the RPS to be consider in the final tally.

3.1 Pre-ballot generation

Before the voting process, the methods and parameters of the election are agreed upon and made public for everyone to consult. Let s denote the private key, in a RSA scheme, only known by the IA. Let n and v denote the public key and modulus components of the same key. Let h be an agreed hash function that produces T_H bit length outputs. Let T_S denote the number of bits in the binary representation of n.

Before interacting with any of the entities, the elector can independently craft his pre-ballot by performing the following steps:

1. The elector selects the desired *choice* for the election.
2. The elector also selects a random (secret) integer invertible modulo n. This value will be the mask used to blind the pre-ballot.
3. She computes the $hash = h(choice \| mask)$.
4. Finally the pre-ballot is crafted by applying the public key v: $(choice \| hash) \cdot mask^v \bmod n$.

3.2 Identification step

Once the pre-ballot is crafted, the elector needs to engage in an identification process with the IA to obtain a valid and certified ballot. Thanks to the blinding factor of the mask, the direction of the vote is not leaked.

1. The elector sends to the IA the pre-ballot and his identification credentials.
2. The IA checks the credentials against the elector census. If valid, it will sign the pre-ballot with its secret key s:

$$((choice \| hash) \cdot mask^v)^s = (choice \| hash)^s \cdot mask \bmod n$$

   and will send it back to the elector. The IA will reject if the credentials are invalid.

3. The elector gets the signed ballot and she is able to cast a vote.

3.3 Casting the ballot

The elector needs to cast the valid ballot to be considered in the final tally. To do so, she sends the ballot, and the inverse of the mask to the RPS. Please note that none of these values reveal any information about the elector's identity.

1. The elector send to the RPS the ballot $(choice \| hash)^s \cdot mask$ and the inverse of the mask $mask^{-1}$.

2. RPS removes the mask by applying its inverse $(choice\|hash)^s \cdot mask \cdot mask^{-1} \bmod n = (choice\|hash)^s$.
3. RPS decrypts the ballot by applying the public key v:

$$(choice\|hash)^{s^v} \bmod n = choice\|hash$$

4. RPS strips the last T_H bits corresponding to the hash and obtains and tallies the elector *choice*.
5. RPS publishes the *hash* so that electors can achieve universal verifiability.

4 Solidity implementation

In this section, we illustrate how our implementation was planned and built. We cover the technical and design decisions that we encountered during the implementation.

4.1 Ethereum and the EVM

We devote this subsection to provide a glimpse of the Ethereum ecosystem. We only cover the general and crucial aspects for our implementation. An experienced reader might want to skip this section. An interested reader can find a detailed and gentle introduction to Ethereum in (Antonopoulos and Wood, 2018).

As mentioned before, Ethereum can be understood as a global and distributed state machine. The state is defined by a sequence of finalized blocks and new block proposals define the update to the state. Blocks are defined by the set of transactions they contain. Transactions usually define an origin, a destination and a set of transactional data that depends on the transaction type. Blocks are atomic in the sense that, either they are processed and a new state is achieved, or they are not processed and the blockchain state remains the same.

Ethereum yellowpaper (Ethereum, 2022) defines the technical specification of the Ethereum Virtual Machine (EVM). The EVM is in charge of processing transactions and updating the current state. All applications and smart contracts need to be EVM compliant to operate in the Ethereum ecosystem. The EVM can be understood as the core technical specification that handles blockchain state, while.

Ethereum is usually employed as a broader term that covers the full ecosystem (e.g.,: client nodes software, external data structures).

The state in Ethereum is handled using an account model (as opposed to the unspent transaction output or UTXO model[2]). This means every address in Ethereum has an attached account balance that can be used to send or receive funds. The global state can be computed from the individual states of each account. There are 2 kind of addresses in Ethereum: Eternally Owned Accounts (EOA) and contract addresses. Users can own EOAs and operate them through digital private keys, signatures and address derivation techniques. Contrarily, contract addresses are solely controlled by the smart contract code. Therefore, contract addresses have no private key associated and all executions are initiated by a transaction coming from a EOA.

As mentioned before, Ethereum uses gas as the unit to measure computational and storage resources. While this gives miners an incentive to run the nodes in the network and prevents malicious uses of the blockchain (e.g.,: an infinite loop), it also forces the users to incur in an expense. Hence, gas optimization techniques are broadly researched and affect the smart contract development. For this reason, Ethereum includes a reduced set of pre-compiled contracts that deal with especially expensive and common mathematical operations such as elliptic curve arithmetic, hashes and signature verification. Otherwise the implementation of this operations would be prohibitively expensive.

4.2 From ECC to RSA

Because of its efficient computation, Ethereum is based, as well as Bitcoin, in the Secp256k1[3] elliptic curve. It also supports some Bn254[4] curve operations, as pre-compiles, because of its friendly pairing properties for zero-knowledge proofs (Bryan et al., 2016; Groth, 2016). This means that no other cryptographic primitive has native or optimized support. TAVS requires RSA signature verification to operate, so tackle the implementation of the necessary support.

Solidity is a Turing complete language, so it is possible to implement any arbitrary system. However, Solidity is a high-level language and only supports a default word size of 32 bytes, which makes it difficult to implement direct support for big integers (integers that require more than 32 bytes to be represented) required in RSA and many other systems based on modular arithmetic. To handle big integers we employed arrays of bytes and assembly code in Yul[5]. Yul is an intermediate level language that can be compiled to low-level bytecode directly used by the EVM. Yul is used to write assembly code for the EVM that can handle lower level details. The goal of Yul is twofold: allows for more detailed management of memory and can save some gas by optimizing code at a lower level.

To implement TAVS, we adapted the big number library developed by Firo[6] to be compatible with the latest Solidity versions. The developed library gives support for basic arithmetic operations using big integers as well as more complex operations that enable cryptographic primitives: modular exponentiation or computing inverses in a given modulo.

2 Blockonomi—Comparing UTXO vs. Account Based-https://blockonomi.com/utxo-vs-account-based-transaction-models/

3 Standard curve database—Secp256k1 https://neuromancer.sk/std/secg/secp256k1

4 Standard curve database—Bn254 https://neuromancer.sk/std/bn/bn254

5 Yul—Bytecode Language https://docs.soliditylang.org/en/latest/yul.html

6 Github—Solidity Big Number https://github.com/firoorg/solidity-BigNumber

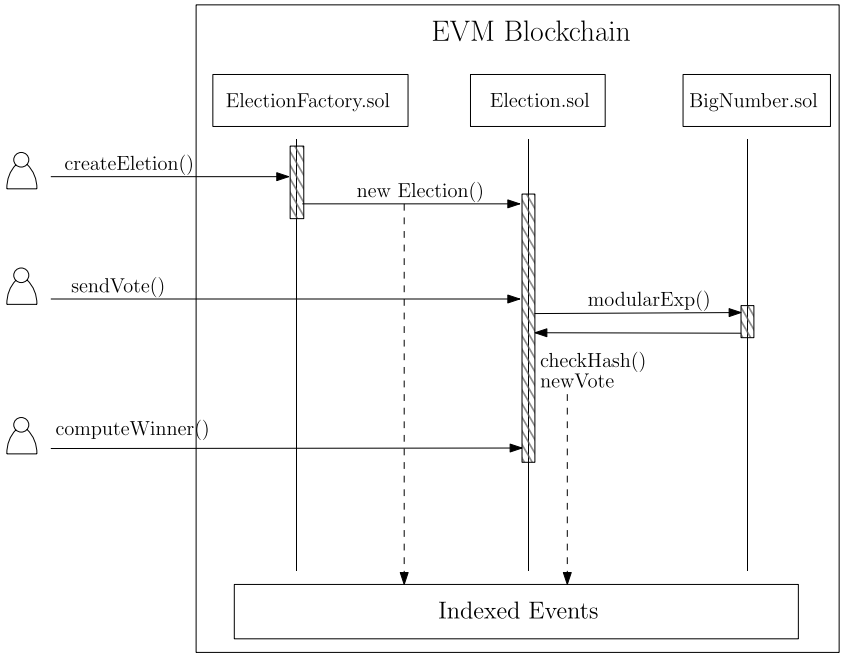

FIGURE 1
Scheme representing the interaction between the user and the smart contracts.

4.3 Events

Being public and being accessible are two different things. Ethereum makes use of events and indexed parameters so the blockchain can be queried more efficiently and integrated with user interfaces. Otherwise, blocks and transactions are long hexadecimal strings that need to be parsed manually.

Events are employed to allow asynchronous triggers with data. User interfaces can be listening for this events which contain return values from EOA initiated transactions. Events also provide a cheaper (in gas) form of storage when compared against smart contract storage. Our implementation contains 2 kind of events: `NewElection` and `NewVote`, so that all the information about the elections carried out can be easily located and traced.

Parameters within events can be indexed (up to 3 of them per event), which provides a finer degree of control for indexing events. So that all events with a given value for a parameter can be filtered from other events with different values.

4.4 Code organization

The Solidity implementation of TAVS is publicly available in Github (https://github.com/Fantoni0/svs). A high level overview of the smart contract interaction can be found in Figure 1. The smart contracts have been tested in a local EVM-compatible network and also have been deployed to a real testnet network. We used Mumbai testnet (a testnet to Polygon) because of the high cost of deploying directly to the main Ethereum network.

All the code has been developed using Solidity version 8.10[7] and the development library Hardhat[8]. Hardhat is one of the most complete Solidity libraries, it allows for compiling and running Solidity code locally, as well as it provides multiple helper functions for debugging smart contracts. The code is structured as follows:

- `contracts`: Contains all the smart contracts that constitute the implementation of TAVS. They are smart contracts written in Solidity.
  - `BigNumber.sol`: A Solidity library adapted from the implementation by Fire[9]. Contains all the code needed to deal with big integers and implement the blind signature scheme based on RSA.
  - `ElectionFactory.sol`: A smart contract that implements the factory design pattern. It creates and deploys instances of Election contracts. Handles and archives the created elections. It emits a event `NewElection` when a new Election is created. The Mumbai network deployment of this contract can be found in address 0xEBa9F87654171f88004f519CC18EfBD8A02e9421.
  - `Election.sol`: It contains all the logic to implement a TAVS election. Handles candidates, verifies votes and stores the current state of the election. It emits a `NewVote` event

7 https://docs.soliditylang.org/en/v0.8.10/
8 https://hardhat.org/
9 Solidity Big Number-https://github.com/firoorg/solidity-BigNumber

when a new Vote is received. The Mumbai network deployment of this contract can be found in address 0x9E459651D2A14B100a310FDd542954bd9565dFC0.

- `deploy`: Contains files to deploy the contracts in networks outside local deployment.
  – `Deploy.ts`: Deploys the ElectionFactory smart contract to a specified network. It creates an Election and sends 10 random votes.
  – `*Template.ts`: A set of template files the interested reader can use to carry out his own election process. See Section 5 for a detailed guide.
- `verify`: Contains the file `arguments.js`, which contains a description of the parameters of the Election smart contract constructor needed to verify the contract. Verifying contracts allows to upload the source code to blockchain explorers such as Etherscan.
- `scripts`: Auxiliary files that implement different functionalities.
  – `Tavs.ts`: Utility functions to simulate the IA in TAVS. it also generates random and valid votes for the scheme.
  – `Utils.js`: Different functions needed for testing: packing parameters, list functions etc.
- `test`: Contains typescript tests to verify the integrity of the smart contracts. See Section 4.5 for more details.

4.5 Tests

Ethereum is a global adversarial network with economic incentives. This means any bug in the code will be public and people will exploit it in order to obtain an economic reward. In addition to this, these can of hacks do not require physical access and allow the attacker to remain anonymous. Hence, testing code is especially relevant when developing smart contracts. The folder `test` contains various tests that analyze the behavior of the code. To run them and verify the validity of the code, the user can simply run `npx hardhat test`. We present here a short list of these assessments and its expected result.

- `ElectionFactory.ts`.
  1. It creates a valid election. It should create a new election and trigger the `NewElection` event.
  2. It creates an invalid election. The test should fail and the transaction reverts with error message "No elections shorter than 1 h allowed".
  3. It creates an invalid election. The test should fail and the transaction reverts with error message "No elections longer than 4 days allowed".
- `Election.ts`.
  1. It creates a valid vote. It should create and send a new vote and trigger the `NewVote` event.
  2. It creates an invalid vote with the wrong hash. The test should fail and the transaction reverts with error message "Invalid hash".
  3. It sends a vote after the election is finished. The test should fail and the transaction reverted with error message "Election has already finished. No more votes accepted".
  4. It should compute the winner of an election. The test sends a unique vote, forces the election to end and checks the winner is the voted candidate.
  5. It tries to compute the winner of an election before the election is finished. The test should fail and the transaction reverted with error message "Election must be finished to compute tally".
  6. It simulates and election with multiple votes and then computes the winner of the election.

4.6 Properties

Our Solidity implementation fulfills all the e-voting properties that TAVS presents. The implementations does not degrade the quality of the voting system. Indeed, some properties benefit from the immutability and decentralization of our approach. We now briefly cover the properties of the TAVS election scheme, and explain how our implementation also accomplishes them. For the actual demonstration of the properties, we refer the reader to the original article (Larriba et al., 2020), where properties are enunciated in Section 4. In this section we consider the arguments in that article and formalize the properties of our implementation in the form of Lemmas. Note that we do not proof the lemma whenever the proof is direct consecuence of the mentioned arguments.

Lemma 4.1. *TAVS is private, and therefore, it is not possible to relate a vote with the elector who casted it.*

Our implementation not only guarantees voters privacys, it also improves the degree of privacy provided in TAVS. TAVS' privacy is derived from the assumption of two honest non-colliding entities. Since we substitute the RPS with a smart contract, this assumption is no longer needed. At first, the idea of privacy and a public blockchain may seem conflicting. Non-etheless, please note that ballots sent to the RPS (the smart contract in our implementation), are not linked to the elector's identity in any form. Once the ballot has been signed by the IA, the elector can generate a burner address, not linked to her in any way, and send the ballot. The validity of the ballot depends on the digital signature by the IA. Hence, we can benefit from the public verifiability of the blockchain without degrading privacy.

Lemma 4.2. *TAVS guarantees the Integrity of the vote, thus it is unfeasible for any partner in the system to modify a ballot without detecting the forgery.*

Lemma 4.3. *TAVS ensures the Correctness of the final tally since it only considers verified and correct ballots.*

Lemma 4.4. *TAVS provides Verifiability since any elector in the census can verify that her vote has been taken into account in the way it was casted.*

Verifiability is hold when all the processing on ballot in order to go through the process do not affect to the ballot itself. In the implementation we propose, the logic of the processing is described in a smart contract, and the results are published in a globally distributed network with thousands of participants. Thus, the implementation of the protocol is more resilient to attacks (as,

TABLE 1 Average costs of execution in the Mumbai and Ethereum network. Average price per gas unit of 43 gwei.

Contract	Method	Gas Units	USD Cost (Mumbai)	USD Cost (Ethereum)
Election	computeWinner	68,000	0.0024$	4.22$
Election	sendVote	39,717	0.015$	2.47$
Election factory	createElection	235,067	0.01$	14.56$
Election factory	Deployment	4,209,467	0.16$	260.81$

for instance, DDoS attacks) than a clasical implementation that considers a private-access public bulletin board. Therefore, the verifiability process of the ballots and their integrity can be publicly audited, guaranteeing that only correct votes are accepted.

Lemma 4.5. *TAVS is a Democratic scheme since only electors in the census can vote.*

Lemma 4.6. *TAVS guarantees the Uniqueness of the votes by ensuring that electors can only vote once.*

Because our implementation maintains the IA and the blind signature scheme, we provide the same democracy and uniqueness as TAVS. The registration procedure does not differ from TAVS. Hence, registered electors can only vote once, since they only have one signed ballot, and, only electors in the census are able to get the signed ballot.

The only difference with respect to TAVS is the immediateness of the tally. In TAVS the final tally is computed and only revealed at the end of the election, since it depends on the authority that plays the role of Remote Polling Station. In our implementation, everything is in the blockchain, hence it is public by default. This means that everyone can see partial votes and compute a partial tally even before the election is finished.

Despite this can be considered not as an issue, it is usually considered as one. To address this (potential) drawback, we note that votes can be encrypted within a public key cryptosystem, whose key is set before hand by the IA (or alternatively by a set of parties), broadcasting the public key together with the election parameters, and revealing the decrypting key only after the end of the election. If in some scenarios the IA cannot be trusted with guarding this key. In those cases, a threshold system could be employed, in such a way that the responsibility of aggregating the key, once the election is finished, rests on a set of reputed parties. For that end, a threshold RSA system (Rabin, 1998; Damgård and Koprowski, 2001) can be used. The parties guarding the keys must hold two requirements: they have to be interested in the correct development of the election; and, they must have antagonistic interests. The first ensures the honest participation of the parties, and the second one prevents malicious collaborations between them. Hence, political parties and/or a subset of electors could conform a suitable the set of guarding parties. This threshold scheme can be configured on demand to tolerate potential errors in the key recovering phase. So that if some parties are unable or unwilling to participate, a subset of honest parties can still decrypt the votes and carry on with the election. Please note, that this would only ensure the anonymity of the tally until the election ends. Furthermore, the privacy of users would not be affected in any form.

5 How to create your own election

In this section we show how to leverage our implementation and the open sourced code to create a particular election. The deployment and interaction will be created in the Mumbai network to reduce gas fees. This tutorial assumes the reader to have an up to date `Node.js` version installed.

1. Clone and install the code.

   git clone https://github.com/Fantoni0/svs
   cd svs/
   npm install
   Compile and verify the smart contracts:
   npx hardhat test

2. Get an address and funds.

   To operate in the blockchain environment it is necessary to have an EOA address. Ownership of addresses is determined by the associated secret key. It is possible to use services such as Vanity-Eth[10] or Crypterium[11] to generate your own key. Once the key is available, it is necessary to add it in the `env/.env` file as `MUMBAI_PRIVATE_KEY`. The next step is to uncomment the lines adding the Mumbai network in `hardhat.config.ts` file. This will let hardhat use the key to send the transactions.

   Once the key is ready, it is mandatory to get some funds available to pay for the transactions gas. Polygon Faucet[12] is a service that gives away small amounts to allow developers to pay for some transactions. To do so it is necessary to paste our address and then claim the funds.

3. Generate the keys to simulate the IA.

   The open-source implementation already includes a pair of public and private RSA keys that simulate the IA. This is sufficient for test purposes but a new pair will be needed to generate secure elections. It is possible to do it using OpenSSL:
   openssl genrsa -out private-key.pem 2048
   openssl rsa -in private-key.pem -pubout -out public-key.pem

10 https://vanity-eth.tk/
11 https://mycrypto.tools/ethaddress.html
12 https://faucet.polygon.technology/

TABLE 2 Costs for computeWinner method. Average price per gas unit of 43 gwei.

Number of candidates	Gas units	USD cost (Mumbai)	USD cost (Ethereum)
2	117,179	0.004$	7.10$
4	129,393	0.005$	7.84$
8	167,397	0.006$	10.14$
16	195,889	0.007$	11.87$
32	307,184	0.011$	18.62$

From the output it is possible to extract the hexadecimal representation from the generated keys and substitute the modulo, public and private key instances in the code.

4. Create a particular Election Factory. (Optional)

There is already an instance of the Election Factory, but a new one can be deployed by running:

npx hardhat deploy --tags ElectionFactory --network mumbai

If generated, the new address must be copied in the smart contract to be deployed. We need the Election Factory address to be able to interact with the contract and create new elections in the future.

5. Create an instance of Election.

It is possible to make use of the file:

`deploy/electionTemplate.ts`

and make the necessary changes to adjust the election to our needs and run:

npx hardhat deploy --tags Election --network mumbai

As done previously, we will need to copy the address in which the Election smart contract was deployed. We'll need the address to send the future votes.

6. Vote.

To finally send a vote, simply set the vote in `deploy/electionTemplate.ts` and execute:

npx hardhat deploy --tags Vote --network mumbai

7. Verify your contract. (Optional)

If it is desired to verify the deployed contracts, we need to obtain some API keys. First the `etherscan_api_key` in `hardhat.config.ts` must be specified. Second, the ElectionFactory program can be simply verified by providing the address in which it was deployed.

npx hardhat verify ADDRESS_TO_VERIFY --network mumbai

To verify the Election contract, since it was called with arguments, it is mandatory to provide the exact same arguments in the deploy/arguments.js, otherwise the verification will fail. To check for the parameters is possible to use the Mumbai Block Explorer.

npx hardhat verify --constructor-args verify/arguments.js --network mumbai ADDRESS_TO_VERIFY

5.1 Gas analysis: Costs of having an election

In this section, we present an empirical study of the gas cost of deploying your own contracts. As mentioned before, gas was introduced in Ethereum as a computation unit to measure the cost associated with specific operations (e.g.,: arithmetic operations, calls, deployments ...). Gas fees are paid in Ethereum's native currency, ether (ETH), and are denoted in gweis. A gwei is equivalent to 10^{-9} ETH. For the rest of this section, we assume the current prices of ETH (1,410$) and MATIC (0.90$) at the moment of writing.

Fees are paid in the currency of the network. Hence, if we operate in the Mumbai testnet, fees will be paid in MATIC, and in ETH if the deployment is made in the Ethereum mainnet. In Table 1 we depict the gas units and USD costs of running the contracts and interacting with them in the Mumbai and Ethereum network respectively. The experiments have been carried out 200 times and the results have been averaged. Table represents the single execution cost of a given method.

All the tests have been run with the same `Hardhat` framework used for development. The plugin.

`hardhat-gas-reporter` provides a detailed report on the gas metrics used in the tests. As covered in Section 4.5, by executing `npx hardhat test`, the user can test the implementation itself, and replicate the presented gas cost analysis. Small variances depending on the average price per gas unit may occur.

As we can appreciate in Table 1, the costs of running the election process in Mumbai is significantly cheaper than carrying out the same process in Ethereum. The methods, no matter the network, show a linear dependency with its complexity. The more computation and storage required, the higher the costs. The deployment of the ElectionFactory contract is specially costly since it needs to upload the whole contract into the blockchain.

All methods exposed in Table 1 are constant in their costs except `computeWinner`. This extrinsic method depends on the internal state of the contract, and needs to iterate over the list of all candidates to compute the most voted one. The length of the candidate list affects the computation, and therefore the gas units and associated costs. Since the previous table only reports average

units over the run of multiple tests, the effect of the candidate list on the `computeWinner` method is not captured. For this reason, we provide a specific description of the cost depending on the cardinality of the candidate's list in Table 2. The cost of calling the method has been isolated from other tests to avoid any possible cross contamination of results.

As we can see in Table 2, while the costs of computing the winner of the election increase with the number of candidates, it is not a severe change. Specially if we consider that elections with 32 or more candidates are unlikely, or we compare the number of gas units with the deployment of the smart contract reflected in Table 1.

Results show that our implementation of TAVS is perfectly feasible and affordable even for large elections. The deployment in networks such as Mumbai, is within everyone's reach, and the use of Ethereum, while more expensive, it is still an effective solution. Specially if we compare the costs with those related to running a traditional election.

6 Conclusion

In this paper we presented our Solidity implementation of the TAVS voting protocol. We showed that, by replacing one the authorities with a smart contract, the overall privacy can be improved by reducing the set of security assumptions. Other properties also benefit from the publicity and auditability properties of the blockchain technology.

We also open-sourced and deployed the implementation to a public testnet, so that users and interested reader can verify and interact with the code. They can benefit from the implementation itself and from the libraries and utilities we developed and adapted to implement and RSA blind signature scheme within the EVM. We further presented a use case to prove the feasibility of our implementation in a real scenario.

As future work, we are working on the code audit and the development of a friendly front-end for the unexperienced user.

Data availability statement

Publicly available datasets were analyzed in this study. This data can be found here: https://github.com/Fantoni0/svs.

Author contributions

Both authors designed the protocol and the smart contract. AL implemented the application, carried out the deployment of the smart contract, and wrote the manuscript. DL wrote and revised the manuscript, as well as coordinated the work.

Acknowledgments

AL would like to thank Eladio Sánchez Checa, who, as an experienced Solidity developer, gave his most friendly help with Solidity. AL also thanks the open source resources that both OpenZeppelin and CryptoZombies make available to new developers.

Conflict of interest

The authors declare that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.

Publisher's note

All claims expressed in this article are solely those of the authors and do not necessarily represent those of their affiliated organizations, or those of the publisher, the editors and the reviewers. Any product that may be evaluated in this article, or claim that may be made by its manufacturer, is not guaranteed or endorsed by the publisher.

References

Antonopoulos, A. M., and Wood, G. (2018). *Mastering Ethereum*. Sebastopol, CA: Building smart contracts and DApps O'Reilly Media Inc.

Bryan, P., Howell, J., Craig, G., and Raykova, M. (2016). Pinocchio: Nearly practical verifiable computation. *Commun. ACM* 59 (2), 103–112. doi:10.1145/2856449

Chaum, D. (1983). "Blind signatures for untraceable payments," in *Advances in cryptology* (Springer), 199–203.

Chouhan, V., and Arora, A. (2022). Blockchain-based secure and transparent election and vote counting mechanism using secret sharing scheme. *J. Ambient Intell. Humaniz. Comput.* (1–19). doi:10.1007/s12652-022-04108-0

Curran, K. (2018). E-voting on the blockchain. *J. Br. Blockchain Assoc.* 1 (2), 4451–4456. doi:10.31585/jbba-1-2-(3)2018

Damgård, I., and Koprowski, M. (2001). "Practical threshold rsa signatures without a trusted dealer," in International conference on the theory and applications of cryptographic techniques (Springer), 152–165.

ElGamal, T. (1985). A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Trans. Inf. theory* 31 (4), 469–472. doi:10.1109/tit.1985.1057074

Ethereum, W. (2014). Ethereum whitepaper. *Ethereum*.

Ethereum, W. (2022). Ethereum yellowpaper. *Ethereum*.

Gao, S., Dong, Z., Guo, R., Jing, C., and Hu, C. (2019). An anti-quantum e-voting protocol in blockchain with audit function. *IEEE Access* 7, 115304–115316. doi:10.1109/access.2019.2935895

Gioulis, A., and Markantonakis, K. (2018). "E-voting with blockchain: An e-voting protocol with decentralisation and voter privacy–1567," in IEEE international conference on internet of things (iThings) and IEEE green computing and communications (GreenCom) and IEEE cyber, physical and social computing (CPSCom) and IEEE smart data (SmartData) (IEEE), 1561.

Goldwasser, S., Micali, S., and Rackoff, C. (2019). "The knowledge complexity of interactive proof-systems," in Providing sound foundations for cryptography: On the work of shafi goldwasser and silvio micali, 203–225.

Goldwasser, S., Micali, S., and Rackoff, C. (1989). The knowledge complexity of interactive proof systems. *SIAM J. Comput.* 18 (1), 186–208. doi:10.1137/0218012

Groth, J. (2016). "On the size of pairing-based non-interactive arguments," in Annual international conference on the theory and applications of cryptographic techniques (Springer), 305–326.

Hreiðarsson, K., Hamdaqa, M., and Hjálmtysson, G. (2018). "Blockchain-based e-voting system," in IEEE 11th international conference on cloud computing (CLOUD) (IEEE), 983–986.

Kshetri, N., and Voas, J. (2018). Blockchain-enabled e-voting. *Ieee Softw.* 35 (4), 95–99. doi:10.1109/ms.2018.2801546

Larriba, A. M., Sempere, J. M., and López, D. (2020). A two authorities electronic vote scheme. *Comput. Secur.* 97, 101940. doi:10.1016/j.cose.2020.101940

Moore, C., O'Neill, M., O'Sullivan, E., Doröz, Y., and Sunar, B. (2014). "Practical homomorphic encryption: A survey," in IEEE international symposium on circuits and systems (ISCAS) (IEEE), 2792–2795.

Moura, T., and Gomes, A. (2017). "Blockchain voting and its effects on election transparency and voter confidence," in Proceedings of the 18th annual international conference on digital government research, 574–575.

Mursi, M. F. M., GhazyAssassa, M. R., Ahmed, A., and Samra, K. M. A. (2013). On the development of electronic voting: A survey. *Int. J. Comput. Appl.* 61 (16), 1–11. doi:10.5120/10009-4872

Nakamoto, S. (2008). Bitcoin whitepaper. *A public-key cryptosystem*.

Niederreiter, H. (1985). "A public-key cryptosystem based on shift register sequences," in Workshop on the theory and application of of cryptographic techniques (Springer), 35–39.

Onur, C., and Arda, Y. (2022). Electanon: A blockchain-based, anonymous, robust and scalable ranked-choice voting protocol. Available at: https://arxiv.org/abs/2204.00057.

Rabin, T. (1998). "A simplified approach to threshold and proactive rsa," in Annual international cryptology conference (Springer), 89–104.

Rivest, R. L., Shamir, A., and Adleman, L. M. (1983). Cryptographic communications system and method, September 20 1983. *U. S. Pat.* 4, 405.

Rivest, R. L., Shamir, A., and Tauman, Y. (2006). "How to leak a secret: Theory and applications of ring signatures," in *Theoretical Computer Science* (Springer), 164–186.

Shamir, A. (1979). How to share a secret. *Commun. ACM* 22 (11), 612–613. doi:10.1145/359168.359176

Taş, R. (2020). A systematic review of challenges and opportunities of blockchain for e-voting. *Symmetry* 12 (8), 1328. doi:10.3390/sym12081328

Yang, X., Yi, X., Nepal, S., Kelarev, A., and Han, F. (2020). Blockchain voting: Publicly verifiable online voting protocol without trusted tallying authorities. *Future Gener. comput. Syst.* 112, 859–874. doi:10.1016/j.future.2020.06.051