Anonymous Voting using Zero-knowledge Proofs

Chinmay Sonar, Rakshith Gopalakrishna, Radha Kumaran

1 Abstract

In this report we propose an anonymous voting protocol that conceals the vote of each participant from all other participants (except the contestant voted for), and design an e-voting system that implements this protocol for leader elections. A public key encryption system is used to ensure anonymity of votes, and a zero-knowledge proof system is used to ensure verifiability of results. Our system is compared against a non-anonymous baseline, to determine the costs involved in ensuring anonymity and verifiability. We also propose an alternate solution that uses a homomorphic encryption system and guarantees more anonymity, but do not discuss any implementation details.

2 Introduction

Traditionally, voting systems require trust in a single party who is responsible for counting votes and declaring the result of the election, and also for ensuring correctness of votes cast. Also, if voters' identities and the votes they cast are made public, voters may be influenced to change their votes if they know they have partial or complete knowledge of distribution of the cast votes among contestants, as well as if the contestants know the identities of the voters who voted for them [1]. Thus anonymity of the voters plays an important part in the election process, allowing for a fair and unbiased election.

There has been work that explores the application of zero knowledge to anonymous voting systems [2] [3], which we extend with a novel approach.

We propose two solutions, both variations of the same general idea. In each round, every voter will `Encrypt` her vote using a specific public key. Next, she'll broadcast a vote on a tamper free broadcast channel. We now have a vote aggregation phase (conducted by an *aggregator*), followed by the winner determination. To check the validity of the election outcome, any participant can verify the zero-knowledge proofs provided during the protocol. Note that we assume an honest majority of participants in the system. The burden of ensuring correctness of the protocol and concealing the voting information is carried by zero-knowledge proofs.

For most of this report we discuss our first solution (protocol) in details both conceptually and in terms of system implementation and deployment. Here, we use a public key encryption scheme with a zero-knowledge proof system. We note that this solution is the more efficient of the two.

Our second solution involves slightly heavy machinery including homomorphic computation along with zero knowledge proofs but it achieves stronger security guarantees, i.e., in particular, here, the voter identities are concealed from both the other voters and the candidate they have voted for. The second solution is merely a conceptual contribution and we do not provide implementation details.

3 Problem Definition

Our aim is to design an anonymous leader election protocol during distributed consensus where we want to conceal votes of the participants. The main entities in our system are the set of participants (voters) $V = \{V_1, V_2, \ldots, V_n\}$. For a given round, let $P \subseteq V$ such that $|P| = \ell \leq n$ be the set of contestants for the round, and t be an integer indicating the threshold. In our proposed approaches, P is known beforehand, and cannot be changed during the course of the round.

Let v_i be the vote of participant V_i for the current round. At a high level, v_i contains the contestant preference for participant V_i. The exact structure of v_i depends on the specific protocol under consideration, in particular, v_i is a single-bit in our approach 1 whilst it is an integer vector of length ℓ in Approach 2.

Problem Statement: Given as input the set of participants V, a set of contestants P, and t we want to design a *fair* leader threshold election protocol such that final leader is computed as a function of votes v_i, and participants are not aware of each other's votes.

Here, fairness means that no voter has any knowledge of the decisions of other voters until the results are obtained.

Next, we turn to the details of our threat model.

3.1 Threat model:

- We consider the active setting where the adversary is malicious and can arbitrarily deviate from the protocol. Out of the n agents in the system we require an honest majority of the nodes.
- Malicious agents can vote multiple times, impersonate other participants, incorrectly aggregate the votes, and lie about the outcome of the election.
- We assume no collusion among any of the agents in the system.
- We assume the public broadcast channel is tamper-free.
- Each agent in the system can be one or more of the following: a voter, a contestant or an aggregator.

4 Solution

In this section, for the most part, we discuss the details of our first approach which is the main technical contribution of this work. We start with an overview of the protocol, and later fill the details for each step. Near the end of this section, we describe our second approach which gives stronger anonymity guarantees.

4.1 Approach 1:

Overview: We refer the reader to Figure 1.

1. **Setup:** Initially, each participant runs a one time $\texttt{KeyGen}(1^\lambda)$ procedure to produce a pair of public and private keys (pk_i, sk_i) where λ is the known security parameter, and broadcast p_k.

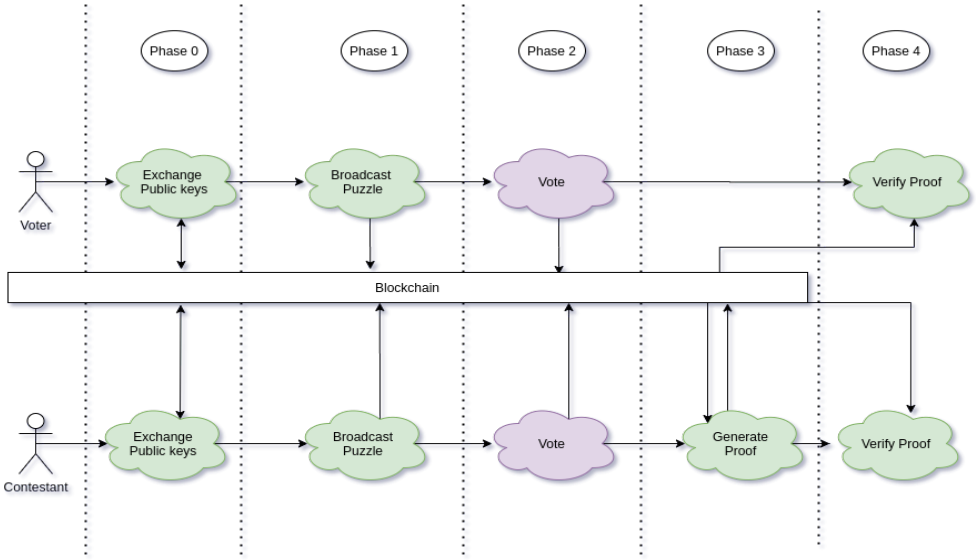


Figure 1: The figure above shows a pictorial overview of protocol 1. Our protocol consists of 5 phases including the setup in phase 0. Here, purple colored phase denotes encrypted messages.

We note that public broadcast channel has properties similar to the blockchain (tamper free evidence log).

2. Broadcast puzzle (voters announcing their participation for this round)
3. Broadcast encrypted vote on the blockchain
4. Contestants gather the votes, and generate and broadcast a zero-knowledge proof.
5. All participants verify all proofs to decide winners.

Overall, this approach uses a simple public key encryption scheme along with generated zero-knowledge proofs. Notice that, during the protocol, voters do not know the votes of the participants. The only information revealed is that each contestant gets to know the set of participants that voted for her. Next, we give further details of our protocol to achieve step 4 which is a critical step of generating zero-knowledge proofs.

Detailed description leading to zero-knowledge proofs: The idea of our zero-knowledge proofs is as follows.

Starting from a solution, each voter produces a puzzle such that finding a solution given only the puzzle is computationally expensive. Each voter broadcasts their puzzle. We assume that the puzzles are compatible in the sense that all n puzzles can be combined to form a final puzzle for the round. To vote, the voter encrypts the solution she knows with the public key of the contestant of her choice. We determine the winner using the threshold value, i.e., a contestant wins only if she receives at least t votes. Contestants decrypt the solutions, and produce a zero-knowledge proof if she has more than t solutions.

Here, the aim of zero-knowledge proofs is to verify that the contestant has received at least t votes without revealing the witness solutions. This is because if the witness solutions are revealed,

then participants can check which puzzle each solution solves and know the identities of the voters voted for the contestant.

We now give further details of our protocol including the puzzle we choose for our implementation.

We use the RSA problem as our puzzle, and obtain the following protocol:

- **Phase 0 (Setup):** In this phase, we send messages without signatures as participants do not have a way to verify the signatures.
- **Phase 1 (Announcement):** All participants sample a pair of large prime number (p_1, p_2) and *publish the product* $n_i = p_1 \times p_2$. Contestants additionally broadcast a message to *declare candidacy*. All messages sent in this and all subsequent phases include digital signatures. Participants discard the messages without verified signature.
- **Phase 2 (Voting):** All participants encrypt their pair of factors using public key of the contestant they want to vote for and broadcast on the blockchain.
- **Phase 3 (Aggregation and Proof generation):** All participants try to to decrypt all the messages sent during voting on the blockchain. For the correctly decrypted messages, they store the pair of factors received. If a participant collects more than t pairs, she generates proof and publish on the blockchain. Contestants gather all public and private inputs for proof generation from the blockchain.
- **Phase 4 (Verification):** All participants collect proofs from phase 3 and the rest of the public inputs from phase 2, and verify all the proofs. During verification, participant computes the public input N which is the product of all broadcast pairs and the total number of voters, and checks for:
  1. Threshold many unique pairs of factors of N
  2. No trivial factor

We assign a pre-defined time window for each of the above phase, and messages broadcast out of phase are discarded. We simulate the time window by restricting each phase to a contiguous range block numbers in the blockchain, which are uniform across all participants.

In the first phase of the above protocol, each participant makes sure that the number they broadcast is not broadcast by any other participant. We give implementation details in Section 5. Next, we explain how our protocol handles various possible actions of malicious participants.

4.1.1 Handling malicious behaviour:

Recall that we consider active setting, i.e., a malicious participant can deviate from the protocol arbitrarily.

- **Malicious Voters:**
  1. Broadcast product multiple (phase 1) or vote (phase 2) multiple times: Honest participants discount n_i sent by such participants during final public input computation which essentially implies that the protocol does not consider the votes of such participants.

2. Verify false proof: Honest majority will overturn any such decision reached by allowing false proofs.
3. Encrypt incorrect factors/ do not vote (node failure): In the former case, the voter will lose her vote which in sub-optimal for her; in the latter, the protocol will update the total number of voters and proceed without one or more votes. For the latter, we decrease the winning threshold accordingly.
4. Broadcast unencrypted factors: This case amounts to collusion which we do not allow in our model.

- **Malicious Contestants:** Recall that a contestant is also a voter, hence, all of the above cases apply for contestants.

1. Broadcast false proof: The case arises either because the contestant computed N incorrectly or if contestant produces a proof without enough votes. Due to an honest majority of participants, proof will rejected.

4.2 Approach 2:

1. We run $\texttt{KeyGen}(1^\lambda)$ to generate a common FHE public key for encryption. Additionally, every participant will generate a secret key.
2. To vote, the voter first constructs a vector of votes of length equal to the number of contestants.
3. The voter then encrypts her vote vector under the common election FHE public key and broadcasts it on the common broadcast channel. All voting happens within a pre-defined time window, and votes broadcast outside of this window are ignored.
4. A participant is chosen in a round-robin fashion to aggregate the votes. Aggregation is done by homomorphic addition of the ciphertexts of the vote vectors.
5. An honest majority of the participants is required to decrypt the aggregated votes. To enable this, we make use of a threshold FHE scheme.
6. The voters prove that they voted correctly and the aggregator proves that the aggregation was done correctly using zero-knowledge proofs.

Notice that this protocol gives a total anonymity, i.e., the votes are concealed from all other participants including the contestant who received the vote. Here, the onus of correctness and anonymity is on the correctness of multi-key FHE scheme and zero-knowledge proofs. Similar to the first solution, malicious behaviours are mainly handled by the honest majority.

5 Implementation

We implemented a prototype of the protocol described above. Our prototype is written in Python. The prototype uses Python's cryptography module [4] for encryption, decryption, signature, and verification. The implementation uses RSA-2048 bit keys for all its cryptographic operations. In addition, the prototype uses pysnark [5] to generate and verify zero knowledge proofs. Finally,

Ethereum [6] is used as the tamper-free public broadcast channel. We implement two variants of the protocol. The protocol implementation adds 734 lines of Python on top of existing libraries.

6 Evaluation

The protocol's evaluation answers the following questions:

1. What are the overheads of the protocol relative to a non-secure baseline?
2. What are the costs of the cryptographic techniques employed by the protocol?

We compare our anonymous implementation with the a non-secure baseline which provides no anonymity. Our implementation builds two levels of security on top of the baseline. The first level adds privacy guarantees using public key cryptography, and the second level adds provable security against malicious adversaries using zero-knowledge proofs.

Table 1 shows the individual costs of the cryptographic primitives in the form of microbenchmarks.

We perform our experiments on the following two machines with Intel Core i7-10750H CPU and 16 GB of main memory and Intel Core i7-10710U CPU and 8 GB of main memory respectively. We run an Ethereum private blockchain on each of these two machines and perform a series of experiments to evaluate our protocol and implementation by varying the number of contestants in the election and recording the costs for each of the participants in the network. The prime numbers used in the puzzles are of 16 bits each.

Operation	CPU Time for 10B	CPU Time for 100B
Encryption	43.23	39.33
Decryption	657.6	629.9
Signature	670.84	727.3
Verification	47.76	48.97

Table 1: CPU times of cryptographic operations for various input message sizes. The RSA key size is 2048 bits. Encryption and Decryption use RSA-OAEP scheme. Signature and verification use RSA-PSS scheme

Tables 2 and 3 show the proof generation time. The function used to generate primes uses the Miller-Rabin test with 128 tests.

Figure 2a shows the total time taken by the protocol to complete for each type of participant. Each of the scenarios tested are of the form "X_Y_Z" where X denotes the anonymous (a) and the non-anonymous (na) protocols, Y stands for the number of contestants (1 or 2), and Z for the participant type (voter or contestant). We can see that the anonymous protocol for 1 contestant election takes 146.2 s to complete compared to 131.492 s for the non-anonymous protocol. We attribute the 11.2% increase in the total time to the overheads of various cryptographic operations our protocol performs.

Table 4 shows the total size of the messages sent and received during the protocol for various scenarios.

Size of Prime Number(bits)	Proof Generation Time(ms)
8	62.29
10	58.67
12	60.83
14	56.96
16	64.28

Table 2: Proof generation times for varying sizes of one prime factor, with the other factor being 8-bit and with 10 voters.

Number of Voters	Proof Generation Time(ms)
2	38.13
4	39.87
6	39.04
8	43.05
10	37.62

Table 3: Proof generation times for varying numbers of voters, using 2 8-bit prime factors.

Scenario	Data sent(bytes)	Data received(bytes)
a_2_c	5694	7199
a_2_v	1505	7199
a_1_c	5690	6933
a_1_v	1243	6933
na_2_c	24	48
na_2_v	24	48
na_1_c	24	37
na_1_v	13	37

Table 4: Amount of data sent and received in each scenario

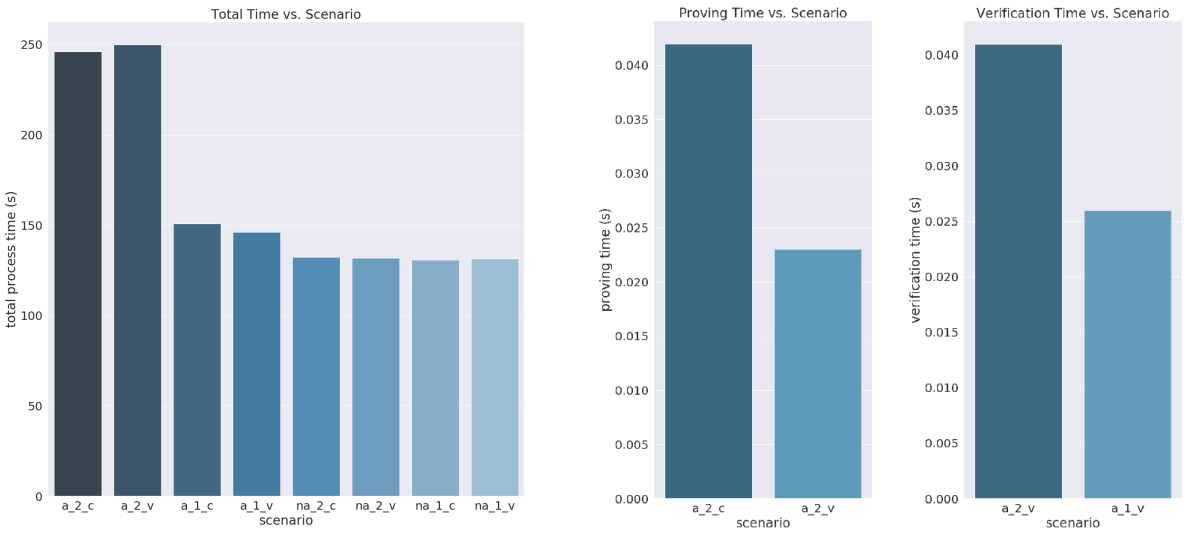
Figure 2b shows the CPU time for a contestant to generate the zero knowledge proof in a two contestant election compared to the CPU time for a voter in the proving phase who does not perform any zero knowledge operations. Proof generation adds 19 ms of CPU time.

Figure 2c shows the CPU time for a verifier in the two contestant election compared to the CPU time for a verifier in the one contestant election. The verifier in the two contestant election needs to verify one additional proof compared to the verifier in the one contestant election which adds 15 ms of overhead.

7 Related Work

Verifiable electronic voting has been around from the start of the century. One of the early work was by Chaum [7] who proposed *Votegrity* which is an end-to-end verifiable scheme. Here, end-to-end verifiability means that the voter can verify that their own vote has been cast as intended. Other protocols which have end-to-end verifiability are [8, 9]. Some of the end-to-end verifiable systems use a trusted public broadcast channel called *bulletin board* [10, 11]. In our approach, we do not require a trusted public broadcast channel as we use blockchain for this purpose.

There has also been work on using blockchain technology to enable transparent and anonymous e-voting. Tarasov and Tewari [2] propose a voting protocol based on blockchain technology, which uses a payment scheme to offer anonymity of transactions. Zcash, a decentralised blockchain payment scheme, provides the required anonymity. Our work aims to provide anonymity without requiring a payment scheme to be implemented.



(a) (b) (c)

Figure 2: (a) Total process time for each scenario tested. (b) Proving time for contestant vs. voter. (c) Verification time with 2 contestants vs. 1 contestant

Hjalmarsson [12] proposes a blockchain based e-voting system with a private-based blockchain, but requires the presence of a trusted party to verify the identity of the voters, which our system does not require.

References

[1] Jac C Heckelman. The effect of the secret ballot on voter turnout rates. *Public Choice*, 82(1-2):107–124, 1995.

[2] Pavel Tarasov and Hitesh Tewari. Internet voting using zcash. *IACR Cryptol. ePrint Arch.*, 2017:585, 2017.

[3] Tassos Dimitriou. Efficient, coercion-free and universally verifiable blockchain-based voting. *Computer Networks*, page 107234, 2020.

[4] pyca. Cryptography. https://pypi.org/project/cryptography/, 2014.

[5] Meilof Veeningen. pysnark. https://github.com/meilof/pysnark/, 2019.

[6] Vitalik Buterin and Gavin Wood. Go-Ethereum. https://github.com/ethereum/go-ethereum, 2015.

[7] David Chaum. Secret-ballot receipts: True voter-verifiable elections. *IEEE security & privacy*, 2(1):38–47, 2004.

[8] Ben Adida. Helios: Web-based open-audit voting. In *USENIX security symposium*, volume 17, pages 335–348, 2008.

[9] Susan Bell, Josh Benaloh, Michael D Byrne, Dana DeBeauvoir, Bryce Eakin, Philip Kortum, Neal McBurnett, Olivier Pereira, Philip B Stark, Dan S Wallach, et al. Star-vote: A secure, transparent, auditable, and reliable voting system. In *2013 Electronic Voting Technology Workshop/Workshop on Trustworthy Elections (EVT/WOTE 13)*, 2013.

[10] James Heather. Implementing stv securely in prêt à voter. In *20th IEEE Computer Security Foundations Symposium (CSF'07)*, pages 157–169. IEEE, 2007.

[11] Arnis Parsovs. Homomorphic tallying for the estonian internet voting system. *IACR Cryptol. ePrint Arch.*, 2016:776, 2016.

[12] Fririk Hjálmarsson, Gunnlaugur K Hreiarsson, Mohammad Hamdaqa, and Gísli Hjálmtỳsson. Blockchain-based e-voting system. In *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, pages 983–986. IEEE, 2018.