Dispute-free Scalable Open Vote Network using zk-SNARKs

Muhammad ElSheikh[1,2] and Amr M. Youssef[1]

[1] Concordia Institute for Information Systems Engineering, Concordia University, Montréal, Québec, Canada
[2] National Institute of Standards (NIS), Cairo, Egypt
{m_elshei,youssef}@ciise.concordia.ca

Abstract. The Open Vote Network is a self-tallying decentralized e-voting protocol suitable for boardroom elections. Currently, it has two Ethereum-based implementations: the first, by McCorry *et al.*, has a scalability issue since all the computations are performed on-chain. The second implementation, by Seifelnasr *et al.*, solves this issue partially by assigning a part of the heavy computations to an off-chain untrusted administrator in a verifiable manner. As a side effect, this second implementation became not dispute-free; there is a need for a tally dispute phase where an observer interrupts the protocol when the administrator cheats, *i.e.,* announces a wrong tally result. In this work, we propose a new smart contract design to tackle the problems in the previous implementations by (i) preforming all the heavy computations off-chain hence achieving higher scalability, and (ii) utilizing zero-knowledge Succinct Non-interactive Argument of Knowledge (zk-SNARK) to verify the correctness of the off-chain computations, hence maintaining the dispute-free property. To demonstrate the effectiveness of our design, we develop prototype implementations on Ethereum and conduct multiple experiments for different implementation options that show a trade-off between the zk-SNARK proof generation time and the smart contract gas cost, including an implementation in which the smart contract consumes a constant amount of gas independent of the number of voters.

Keywords: Open Vote Network · E-voting · Blockchain · zk-SNARK · Smart contracts · Ethereum

1 Introduction

E-voting refers to an election system in which voters can cast their vote electronically. The main advantages of e-voting, compared to the traditional paper-based election, include high speed of tallying, cost-effectiveness, and scalability. Using e-voting systems can be crucial in many situations, *e.g.,* the current COVID-19 pandemic renders traditional paper-based voting within organizations a potential health hazard and sometimes not possible because of the work from home setup. Nowadays, there are many e-voting systems that can support a number of voters from a boardroom to a national scale [6,16,1]. However, most of them rely heavily on a trusted central authority, which might lead to violating the voters' privacy. With the emerging of blockchain technology as a decentralized append-only ledger, many researchers have proposed several blockchain-based e-voting protocols (*e.g.,*, see [13,14,15,17,18,21]). Unfortunately, widely deployed blockchains such as Bitcoin and Ethereum suffer from scalability issues. Moreover, they do not inherently provide the privacy required by e-voting protocols. Therefore, a good blockchain-based e-voting system should handle these limitations.

The Open Vote Network is a self-tallying decentralized voting protocol. Self-tallying means that anyone who observes the protocol can tally the result without counting on a trusted authority. The protocol also provides maximum voter privacy; a single vote can only be breached by a full-collusion involving compromising all other votes. McCorry *et al.* [17] presented the first implementation of the Open Vote Network protocol on the Ethereum blockchain. However, their implementation does not provide scalability because all the protocol computations are delegated to the smart contract. This problem is partially solved by Seifelnasr *et al.* [21] by assigning the tallying computations to an off-chain untrusted administrator in a verifiable manner. To address the possibility of a malicious administrator, they added a dispute phase in which an honest voter may interrupt the protocol if the administrator provides an incorrect tallying result. As a result, the protocol lost its dispute-free property.

Contribution. In this work, we provide a new design to deploy the Open Vote Network using Ethereum smart contract. The new design can achieve better scalability without loosing the dispute-free property. Our contribution can be summarized as follows.

1. We develop a smart contract for the Open Vote Network in which all the heavy computations are performed off-chain without loosing dispute-free property.
2. We design three zk-SNARK arithmetic circuits to verify that all the off-chain computations are performed correctly by their responsible parties.
3. We develop a prototype[3] of our design to assess its performance. We also conduct some experiments to estimate the maximum number of voters that can be supported before exceeding the gas limit of the Ethereum block.
4. Finally, we show how to enhance the scalability of our design by modifying the zk-SNARK circuits such that they have statements of a fixed size. Consequently, the smart contract functions which verify the correctness of these zk-SNARK proofs consume fixed gas cost independent of the number of voters. The tradeoff between the zk-SNARK proof generation time and the smart contract gas cost is also experimentally evaluated.

The rest of the paper is organized as follows. In Section 2, we briefly revisit some related work on voting protocols implemented on the Ethereum blockchain.

[3] https://github.com/mhgharieb/zkSNARK-Open-Vote-Network

Section 3 recalls the cryptographic primitives utilized in our protocol. In Section 4, we provide our design of the zk-SNARK circuits and the smart contracts. In Section 5, we evaluate our design and compare it against previous work. In Section 6, we provide multiple enhancements to the design in order to achieve better scalability, with different trade-offs between proof generation time and gas cost. Finally, our conclusion is presented in Section 7.

2 Related Work

The Open Vote Network is a self-tallying decentralized e-voting protocol. The concept of self-tallying was introduced by Kiayias and Yung [12] for boardroom voting. This work was followed by Groth *et al.* [7] and Hao *et al.* [10] who proposed a system that provides better efficiency for each voter. Hao *et al.*'s protocol has the same security properties and achieves better efficiency in terms of number of rounds. Li *et al.* [15] presented a new blockchain based self-tallying voting protocol for decentralized IoT. Recently, Li *et al.* [14] proposed a self-tallying protocol that utilizes homomorphic time-lock puzzles to encrypt the votes for a specified duration of time to maintain the privacy of ballots during the casting phase.

McCorry *et al.* [17] and Seifelnasr *et al.* [21] presented two implementations of the system of Hao *et al.* [10] as smart contracts on Ethereum. As mentioned above, the first implementation suffers from a scalability issue and the second requires a third-party to observe the behavior of the election administrator.

3 Preliminaries

3.1 zk-SNARK

A zk-SNARK refers to a zero-knowledge Succinct Non-interactive Argument of Knowledge scheme which enables a prover to convince a verifier that a statement is true without prior interactions between them [8].

Suppose an arithmetic circuit C with a relation $\mathcal{R}_C$ and a language $\mathcal{L}_C$ takes as input a statement $\vec{s}$ and a witness $\vec{w}$ s.t. $(\vec{s},\ \vec{w}) \in \mathcal{R}_C$. A zk-SNARK for this arithmetic circuit satisfiability is defined by the following triple of polynomial-time algorithms [8,9,19]:

- $(\mathsf{pk}, \mathsf{vk}) \leftarrow \mathsf{Setup}(1^\lambda,\ C)$. Given a security parameter λ and the circuit C, the algorithm generates a common reference string (CRS) that contains a pair of keys; a proving key pk and a verifying key vk. Both keys are considered as public parameters for the circuit C.
- $\pi \leftarrow \mathsf{Prove}(\mathsf{pk},\ \vec{s},\ \vec{w})$. Given a proving key pk, a statement $\vec{s}$, and a witness $\vec{w}$ s.t. $(\vec{s},\ \vec{w}) \in \mathcal{R}_C$, the algorithm generates a zero-knowledge non-interactive proof π for the statement $\vec{s} \in \mathcal{L}_C$ that reflects the relation between $\vec{s}$ and $\vec{w}$.
- $0/1 \leftarrow \mathsf{Verify}(\mathsf{vk},\ \vec{s},\ \pi)$. Given a verifying key vk, the statement $\vec{s}$, and the proof π, the algorithm outputs 1 if π is a valid proof for the statement $\vec{s} \in \mathcal{L}_C$, and outputs 0 otherwise.

Typically, a zk-SNARK provides the following security properties [9]:

1. **Perfect Completeness**: For each valid statement $\vec{s}$ with a valid witness $\vec{w}$ s.t. $(\vec{s},\ \vec{w}) \in \mathcal{R}_C$, an honest prover always convinces an honest verifier, *i.e.,* $\mathsf{Verify}(\mathsf{vk},\ \vec{s},\ \pi)$ outputs 1 with a probability equal to 1.
2. **Computational Soundness**: A polynomial-time malicious prover cannot convince the verifier of a false statement, *i.e.,* $\mathsf{Verify}(\mathsf{vk},\ \vec{s},\ \pi)$ outputs 1 with a probability ≈ 0 when the statement $\vec{s} \notin \mathcal{L}_C$.
3. **Computational Zero-Knowledge**: A polynomial-time adversary cannot extract any information about the witness from the honestly-generated proof.
4. **Succinctness**. A zk-SNARK is succinct if the honestly-generated proof size is polynomial in λ and $\mathsf{Verify}(\mathsf{vk},\ \vec{s},\ \pi)$ runs in polynomial time in $\lambda + |\vec{s}|$.

3.2 Open Vote Network

The Open Vote Network is a decentralized two-round self-tallying e-voting protocol [10]. It is suitable for a boardroom election in which the number of voters is relatively small.

In the beginning, eligible voters $(\mathcal{P}_0, \mathcal{P}_1, \ldots, \mathcal{P}_{n-1})$ agree on a finite cyclic group $\mathbb{G}$ of a prime order q and a generator g in which the Decisional Diffie-Hellman (DDH) problem is intractable. Then, each voter $\mathcal{P}_i$ picks a random value $x_i \in_R \mathbb{Z}_q$ as her private voting key. The Open Vote Network is executed for an election with two options 1 or 0 (implying 'YES' or 'NO') as follows.

Round 1. Each eligible voter $\mathcal{P}_i$ publishes her public voting key g^{x_i} along with a non-interactive zero-knowledge proof of knowledge regarding her private voting key x_i on the public bulletin board. At the end of this round, each voter verifies the validity of other voters' zero-knowledge proof of knowledge, then computes her blinding key Y_i as in Eq. 1.

$$Y_i = \prod_{j=0}^{i-1} g^{x_j} / \prod_{j=i+1}^{n-1} g^{x_j} \tag{1}$$

By implicitly setting $Y_i = g^{y_i}$, it is easy to prove that $\prod_i Y_i^{x_i} = g^{\sum_i x_i y_i} = g^0 = 1$.

Round 2. Each eligible voter $\mathcal{P}_i$ uses her blinding key Y_i and the private key x_i to encrypt the vote $v_i \in \{0, 1\}$ s.t. the encrypted vote $V_i = g^{v_i} Y_i^{x_i}$. Then she publishes the encrypted vote V_i along with a non-interactive zero-knowledge proof of validity to prove that the encrypted vote V_i is well-formed such that $v_i \in \{0, 1\}$. At the end of this round and after verifying the non-interactive zero-knowledge proofs of all encrypted votes, anyone who observes the protocol can compute the tally of 'YES' votes by exploiting the homomorphic property in the encrypted votes as follows: $\prod_i V_i = \prod_i g^{x_i y_i} g^{v_i} = g^{\sum_i x_i y_i + v_i} = g^{\sum_i v_i}$. Accordingly, the tally result of 'YES' votes $\sum_i v_i$ can be easily obtained by performing an exhaustive search on the discrete log of $g^{\sum_i v_i}$. This exhaustive search is bounded by the number of voters which is relatively small. For more details, see [10].

4 Protocol Design

In this section, we present our proposed design to deploy the Open Vote Network on the Ethereum blockchain using zk-SNARKs.

4.1 zk-SNARK Arithmetic Circuit

Since validating zk-SNARK proofs on Ethereum are performed over two cyclic groups of prime order p [3], our protocol computations are preformed over a cyclic group $\mathbb{G}$ with a finite field $\mathbb{F}_p$ on the elliptic curve *Baby Jubjub* [2]. $\mathbb{G}$ has a prime order q, a base point (generator) G, and a point at infinity (the neutral element) O. A point $P \in \mathbb{G}$ is presented by its two coordinate values (P^x, P^y).

In our design, we use zk-SNARKs to verify that all off-chain computations are performed correctly by their responsible parties. To this end, we design three zk-SNARK arithmetic circuits: publicKeyGen, encryptedVoteGen, and Tallying corresponding to generating the public key of voters, encrypting the votes, and tallying the result of 'YES' option, respectively.

We design these circuits based on Groth16 zk-SNARK construction [9] because it is a quadratic arithmetic program (QAP) hence it provides a linear-time Setup, quasilinear-time Prove, and linear-time Verify [20]. However, Groth16 enforces some restrictions on the design (*e.g.,* array indices and loop iteration counts must be constant during compiling (Setup) phase).

During the design, we use the following pre-defined arithmetic circuits as building blocks:

- Mux(s, P, Q): Returns P if the selector $s = 0$, and Q if $s = 1$.
- LessThan(a, b): Returns 1 if $a < b$, and 0 otherwise.
- GreaterThan(a, b): Returns 1 if $a > b$, and 0 otherwise.
- CompC(a, c) where c is a constant: Returns 1 if $a > c$, and 0 otherwise.
- Bits2Num$(a_0, \ldots, a_{k-1})$: Returns the integer number represented by bits $a_0, \ldots, a_{k-1}$.
- IsPoint(x, y): Returns 1 if the pair (x, y) is a point on the elliptic curve, and 0 otherwise.
- IsEqual$(P,\ Q)$: Returns 1 if the two points P and Q are equal, and 0 otherwise.
- eADD(P, Q): Point addition $(P + Q)$ on the elliptic curve.
- eSUB(P, Q): point subtraction $(P - Q)$ on the elliptic curve.
- eScalarMUL(a, P): Scalar multiplication (aP) on the elliptic curve.

Let $\kappa = |p| - 1$. Since all operations are preformed over $\mathbb{F}_p$, the number of inputs to Bits2Num must be $\leq \kappa$ bits to avoid overflow when calculating the output value.

publicKeyGen Circuit (C_{PK}). This circuit (see Circuit 1) is intended for the setup of the zk-SNARK to prove that a voter B knows the private key x_B corresponding to the public key $PK_B = x_B G$ in step (1); and the sign of its

x-coordinate is 0 (*i.e.,* $pk_B^{x_B} < p/2$), in step (3), to ensure that PK_B follows the compact representation as described in [11] such that x and y coordinates have a one-to-one relation.

encryptedVoteGen Circuit (C_V). This circuit is intended for the setup of the zk-SNARK to prove that a voter B with index i_B forms her encrypted vote V_B correctly s.t. the vote $v_B \in \{0, 1\}$ as shown in Circuit 2.

Since the verification time is linearly proportional to the size of the statement $\vec{s}$, we decompose each public key PK_i into its coordinate values (pk_i^x, pk_i^y). Since they have a one-to-one relation, the y-coordinate becomes a part of the statement $\vec{s}$ and the x-coordinate becomes a part of the witness $\vec{w}$ in order to optimize the circuit and reduce the verification time, hence the on-chain computation.

From Eq. 1, the encrypted vote V_B is computed as follows:

$$V_B = v_B G + x_B Y_B \tag{2}$$

$$Y_B = \underbrace{\sum_{i=0}^{i_B-1} PK_i}_{Y_l} - \underbrace{\sum_{i=i_B+1}^{n-1} PK_i}_{Y_g} \tag{3}$$

Accordingly, encryptedVoteGen circuit checks that $v_B \in \{0, 1\}$ in step (1); each pair (pk_i^x, pk_i^y) is a point on the elliptic curve in step (5); the sign of the x-coordinate is 0 ($pk_B^{x_B} < p/2$) in step (6) to verify the one-to-one relation; the correctness of Y_l, Y_g, and Y_B (as in Eq. 3) in steps (8-10), (11-13), and (15), receptively; and the correctness of V_B (as in Eq. 2) in steps (16-18).

Tallying Circuit (C_T). This circuit is intended for the setup of the zk-SNARK to prove the correctness of the tallying result as shown in Circuit 3. Similar to C_V, each encrypted vote V_i is decomposed into its coordinate values (V_i^x, V_i^y), then the y-coordinate becomes a part of the statement $\vec{s}$ and the x-coordinate becomes a part of the witness $\vec{w}$ in order to reduce the on-chain computation. Since we cannot enforce the compact representation for V_i, we instead present the sign of x-coordinate in a single bit S_i then compress every κ sign-bits in one integer number D_j. Hence, we can verify the one-to-one relation between the two coordinates of V_i.

Accordingly, Tallying circuit checks that each pair (V_i^x, V_i^y) is a point on the elliptic curve in step (6). Then, it checks the correctness of $\sum_i V_i$ in step (7); the x-coordinate sign (S_i) of V_i in step (9); the compression of every κ sign-bits into an integer D_j in step (12); the incrementally exhaustive search in steps (17-19); and finally the tallying result (res) s.t. $\sum V_i = (res)G$ in steps (21-22). It should be mentioned that $0 \leq res \leq n$, where $res = 0$ when no voter selects the 'Yes' option and $res = n$ when all voters select the 'Yes' option. Therefore, the exhaustive search counter i in step (16) starts from 0 to n.

Circuit 1: publicKeyGen

```
Statement $\vec{s}$: $PK_B$
Witness $\vec{w}$: $x_B$
1 $PK_B \leftarrow \mathsf{eScalarMUL}(x_B, G)$
2 $(pk_B^x, pk_B^y) \leftarrow PK_B$
3 Assert $\mathsf{CompC}(pk_B^x, p/2) = 0$
```

Circuit 2: encryptedVoteGen

```
Statement $\vec{s}$: $(V_B,\ i_B,\ \{pk_i^y\})$
Witness $\vec{w}$: $(v_B,\ x_B,\ \{pk_i^x\})$
1 Assert $(1 - v_B) \times v_B = 0$
2 $Y_l \leftarrow O$
3 $Y_g \leftarrow O$
4 for $i \leftarrow 0$ to $n-1$ do
5     Assert $\mathsf{IsPoint}(pk_i^x,\ pk_i^y)$
6     Assert $\mathsf{CompC}(pk_i^x, p/2) = 0$
7     $PK_i \leftarrow (pk_i^x,\ pk_i^y)$
8     $e_l \leftarrow \mathsf{LessThan}(i,\ i_B)$
9     $T_l \leftarrow \mathsf{Mux}(e_l,\ O,\ PK_i)$
10    $Y_l \leftarrow \mathsf{eADD}(Y_l, T_l)$
11    $e_g \leftarrow \mathsf{GreaterThan}(i,\ i_B)$
12    $T_g \leftarrow \mathsf{Mux}(e_g,\ O,\ PK_i)$
13    $Y_g \leftarrow \mathsf{eADD}(Y_g, T_g)$
14 end
15 $Y_B \leftarrow \mathsf{eSUB}(Y_l, Y_g)$
16 $T_0 \leftarrow \mathsf{eScalarMUL}(x_B, Y_B)$
17 $T_1 \leftarrow \mathsf{Mux}(v_B,\ O,\ G)$
18 $V_B \leftarrow \mathsf{eADD}(T_0, T_1)$
```

Circuit 3: Tallying

```
Statement $\vec{s}$: $(res,\{D_j\},\{V_i^y\})$
Witness $\vec{w}$: $\{V_i^x\}$
1 $l \leftarrow \lceil \frac{n}{\kappa} \rceil$
2 $\{S_i \leftarrow 0, 0 \le i \le \kappa l - 1\}$
3 $\{D_j, 0 \le j \le l - 1\}$
4 $sumV \leftarrow O$
5 for $i \leftarrow 0$ to $n-1$ do
6     Assert $\mathsf{IsPoint}(V_i^x,\ V_i^y)$
7     $sumV \leftarrow \mathsf{eADD}(sumV, V_i)$
8     $V_i \leftarrow (V_i^x,\ V_i^y)$
9     $S_i \leftarrow \mathsf{CompC}(V_i^x, p/2)$
10 end
11 for $j \leftarrow 0$ to $l-1$ do
12    $D_j \leftarrow$
        $\mathsf{Bits2Num}(S_{\kappa j}, \ldots, S_{\kappa j+\kappa-1})$
13 end
14 $t \leftarrow 0$
15 $T \leftarrow O$
16 for $i \leftarrow 0$ to $n$ do
17    $e \leftarrow \mathsf{IsEqual}(T, sumV)$
18    $t \leftarrow t + e \times i$
19    $T \leftarrow \mathsf{eADD}(T, G)$
20 end
21 Assert
     $\mathsf{eScalarMUL}(t, G) = sumV$
22 $res \leftarrow t$
```

4.2 Open Vote Network Smart Contract

Before executing the protocol, an administrator A and a set of n eligible voters run an MPC-based setup ceremony for generating the proving and verifying keys for the arithmetic circuits.

$$(\mathsf{pk}_{C_{PK}}, \mathsf{vk}_{C_{PK}}) \leftarrow \mathsf{Setup}(1^\lambda, C_{PK})$$
$$(\mathsf{pk}_{C_V}, \mathsf{vk}_{C_V}) \leftarrow \mathsf{Setup}(1^\lambda, C_V)$$
$$(\mathsf{pk}_{C_T}, \mathsf{vk}_{C_T}) \leftarrow \mathsf{Setup}(1^\lambda, C_T)$$

After that, similar to [21], the administrator accumulates the list of eligible voters in a Merkle tree $MT_{\mathcal{E}}$ where the voters' Ethereum account addresses are the tree leaves, and publishes it on IPFS allowing each voter to generate her proof of voting eligibility. Also, the administrator defines the time intervals of the protocol phases, namely, Registering Voters, Casting Encrypted Votes, Tallying the Result, and Refunding.

Smart Contract Deployment. Subsequently, the administrator deploys the smart contract, which initializes the variables used in the subsequent phases, with the following set of parameters and pays a collateral deposit F as shown in Fig. 1.

```
Deploy:    upon receiving ($root_{\mathcal{E}}$, $\mathsf{vk}_{C_{PK}}$, $\mathsf{vk}_{C_V}$, $\mathsf{vk}_{C_T}$, $T_1$, $T_2$, $T_3$, $T_4$, $n$)
           from administrator A:
           Assert $value = F$
           Set $admin :=$ A
           Store $root_{\mathcal{E}}, \mathsf{vk}_{C_{PK}}, \mathsf{vk}_{C_V}, \mathsf{vk}_{C_T}, T_1, T_2, T_3, T_4, n$
           Init $voters := \{\}$, $publicKeys := \{\}$, $EncryptedVotes := \{\}$,
           $Int\_Vsigns := \{0, \ldots, 0\}$, $tallyingResult :=$ NULL, $index := 0$
```

Fig. 1: Pseudocode for deployment of the smart contract.

- $root_{\mathcal{E}}$: The root of the Merkle tree $MT_{\mathcal{E}}$.
- $\mathsf{vk}_{C_{PK}}, \mathsf{vk}_{C_V}, \mathsf{vk}_{C_T}$: The verifying keys of the zk-SNARK circuits.
- T_1, T_2, T_3, T_4: The block heights that define the end of the protocol phases.
- n: The number of eligible voters.
- F: A collateral deposit paid by both the administrator and the voters to penalize malicious actors.
- $voters$, $publicKeys$, $EncryptedVotes$, Int_Vsigns: Four arrays of sizes n, n, n, and $l = \lceil \frac{n}{\kappa} \rceil$, used to store the voters' Ethereum account addresses, $\{PK_i\}$, $\{V_i\}$, and $\{D_j\}$, respectively.

Registering Voters. This phase starts immediately after deploying the contract. Each voter B selects her private key x_B and uses publicKeyGen circuit along with $\mathsf{pk}_{C_{PK}}$ to generate the public key PK_B and a zk-SNARK proof π_{x_B}. After that, she invokes Register function in the smart contract with the parameters PK_B and π_{x_B} along with a Merkle proof of voters membership π_B, and pays the collateral deposit F as shown in Fig. 2. On its turn, the function ensures that the voter deposits the correct collateral fee, it is invoked within the allowed interval, and the number of already registered voters does not exceed the total number of eligible voters. After that, it reconstructs the zk-SNARK statement $\vec{s}$ of publicKeyGen circuit. Consequently, Register verifies both the Merkle tree proof of the voter membership and the zk-SNARK proof using $\mathsf{vk}_{C_{PK}}$ key. We utilize the implementation in [4] to verify the Merkle proof. Finally, it stores the public key PK_B and the address of the voter B for the subsequent phases.

```
Register:   upon receiving $(PK_B, \pi_{x_B}, \pi_B)$ from voter B:
            Assert $value = F$
            Assert $T < T_1$
            Assert $index < n$
            Reconstruct $\vec{s} := PK_B$
            Assert MerkleTree.verify( $\pi_B$, B, $root_{\mathcal{E}}$)
            Assert zkSNARK.verify($\mathsf{vk}_{C_{PK}}$, $\vec{s}$, $\pi_{x_B}$)
            Store $publicKeys[index] := PK_B$
            Store $voters[index] :=$ B
            Set $index := index + 1$
```

Fig. 2: Pseudocode for Register function

Casting Encrypted Votes. After the voters registration phase ends, each voter B, which has index i_B, starts computing her blinding key Y_B to encrypt her vote v_B and generate a zk-SNARK proof π_{v_B} using encryptVoteGen circuit along with the proving key pk_{C_V}. Then, she publishes the encrypted vote V_B publicly by invoking CastVote function in the smart contract. The function verifies that the voter casts the encrypted vote within the allowed time interval, and the voter indeed has the index i_B. After that, it reconstructs the statement $\vec{s}$ of encryptedVoteGen circuit along with vk_{C_V} in order to verify the correctness of the zk-SNARK proof π_{v_B} submitted by the voter. Finally, it stores the encrypted vote V_B and updates Int_Vsigns based on the sign of its x-coordinate, V_B^x, and the voter index i_B as depicted in Fig. 3.

```
CastVote:   upon receiving $(V_B,\ i_B,\ \pi_{v_B})$ from voter B
            Assert $T_1$ < $T$ < $T_2$
            Assert B = $voters[i_B]$
            Reconstruct $\vec{s} := (V_B,\ i_B,\ publicKeys)$
            Assert zkSNARK.verify($\mathsf{vk}_{C_V}$, $\vec{s}$, $\pi_{v_B}$)
            Set $EncryptedVotes[i_B] := V_B$
            IF($V_B^x > p/2$):
                Set $Int\_Vsigns[\lfloor \frac{i_B}{\kappa} \rfloor] := Int\_Vsigns[\lfloor \frac{i_B}{\kappa} \rfloor] \oplus 2^{i_B \bmod \kappa}$
```

Fig. 3: Pseudocode for CastVote function

Tallying the Result. After the casting phase ends, the administrator obtains all the encrypted votes stored in the smart contract in order to tally the result of 'YES' option. To this end, she uses Tallying circuit along with pk_{C_T} to obtain the result res and its corresponding zk-SNARK proof π_{res}. Then, she invokes SetTally function to publish this result. The function verifies that the transaction is within the allowed time interval and it is sent by the administrator who deployed the smart contract. Then, the function reconstucts the statement $\vec{s}$ of Tallying circuit to verify the correctness of the tallying result as shown in Fig. 4.

```
SetTally:    upon receiving ($result$, $\pi_{res}$) from administrator A:
             Assert $admin$ = A
             Assert $T_2$ < $T$ < $T_3$
             Set $\vec{s} := (result, Int\_Vsigns, EncryptedVotes)$
             Assert zkSNARK.verify(vk$_{C_T}$, $\vec{s}$, $\pi_{res}$)
             Set $tallyingResult := result$
```

Fig. 4: Pseudocode for SetTally function

Refunding. Finally, at the end of tallying the result phase, the administrator and voters invoke Refund function to reclaim the collateral fee F that they deposited during deploying the smart contract and registering voters phases, respectively. Before refunding the value, this function verifies that the time to set the tallying result has ended and the transaction sender indeed paid the deposit.

5 Evaluation and Comparison

We developed a prototype of our design to evaluate the gas cost and its scalability. The prototype is available as open-source on Github[4]. We utilize `circom` (v2.0) library[5] to compile the arithmetic circuits publicKeyGen, encryptedVoteGen, and Tallying. During the protocol execution, we use `snarkjs` (v0.4.10) library[6] to handle the MPC-based setup ceremony for generating the proving and verifying keys. Also, we use it to generate the zk-SNARK proofs. We utilize `Tuffle` framework[7] (v5.4.24) to deploy and test the smart contracts.

The prototype includes three smart contracts: verifierMerkleTreeCon, verifierZKSNARKCon, and eVoteCon. The first two contracts are deployed as generic functions to verify the Merkle tree proof of membership and the zk-SNARKs proofs. Table 1 summarizes the gas units consumed by different functions in the contracts for executing the protocol for 40 voters.

[4] https://github.com/mhgharieb/zkSNARK-Open-Vote-Network
[5] https://docs.circom.io/
[6] https://github.com/iden3/snarkjs
[7] https://trufflesuite.com/

Table 1: The gas cost for functions in the voting contract

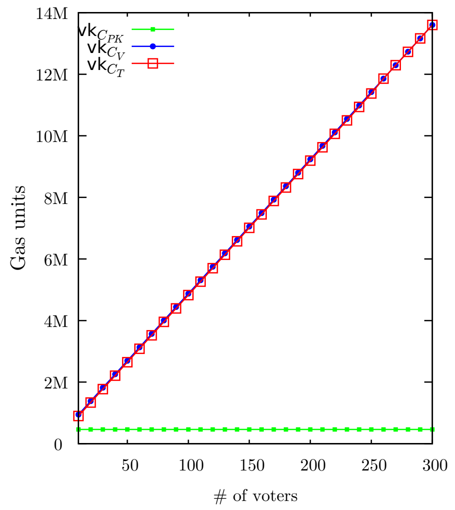
Function	Gas units
verifierMerkleTreeCon	192,013
verifierZKSNARKCon	1,346,155
eVoteCon	1,474,818
`setVerifyingKey`($\mathsf{vk}_{C_{PK}}$)	462,309
`setVerifyingKey`(vk_{C_V})	2,253,285
`setVerifyingKey`(vk_{C_T})	2,209,576
`Register`	349,517
`CastVote`	937,299
`SetTally`	901,532
`Refund`	52,253

Scalability Experiments. We also conduct some experiments to estimate the maximum number of voters who can participate in the same election before exceeding the gas limit of the Ethereum block. To this end, we run the prototype with incremental steps starting from 10 voters up to 300 voters. The gas cost of these experiments is shown in Fig. 5. As depicted in Fig. 5.a, setting the verifying keys vk_{C_V} and vk_{C_T} consumed an amount of gas units linearly with the number of voters. In contrast, the gas cost for setting the verifying key $\mathsf{vk}_{C_{PK}}$ is approximately constant. This behavior is expected since the statement size $|\vec{s}|$ of the circuits publicKeyGen and Tallying is linearly proportional to the number of voters. In contrast, it is constant in the case of encryptedVoteGen circuit. The same behavior is observed for invoking the smart contract functions as shown in Fig. 5.b.

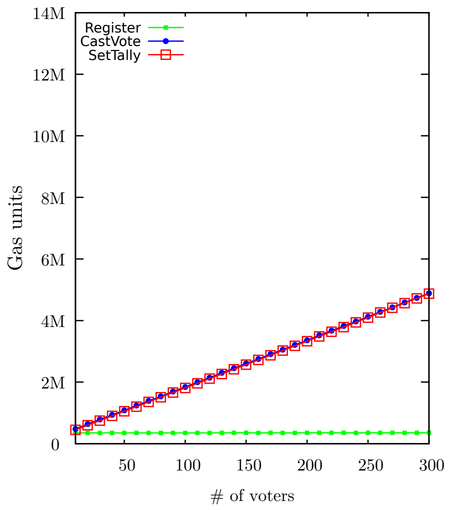
It is obvious that setting a verifying key can be made in multiple blocks. In contrast, running the smart contract function must be completed in the same block. Therefore, the gas cost of invoking the functions is the bottleneck against the scalability. With the current gas limit (30M gas units[8] on Jan. 16, 2022), this prototype can be scaled to around 2000 voters.

Comparison with McCorry et al. [17]. The design in [17] and ours provide the same properties specially the dispute-free property. In our design, we delegate the heavy computation to off-chain parties in a verifiable manner. In contrast, in [17] all the computations are performed on-chain. For the sake of fairness, we compare the gas consumption by each voter and the administrator during a 40-voter election in the two designs as summarized in Table 2. In our design, the gas cost per voter and the administrator are around 40% and 70% of the other design, respectively. However, the gas cost increases rabidly with the number of voters in McCorry *et al.* design as deduced from Fig. 4 in [17]. Therefore, our design is more scalable.

[8] https://ethstats.net/



(a) Gas cost of setting Verification Keys



(b) Gas cost of invoking functions

Fig. 5: Gas cost in scalability experiments

Table 2: Gas cost comparison between our implementation and [17]

Sender	Ours	[17]
Voter	1,339,069	3,323,642
Admin	8,719,141	12,436,190

Comparison with Seifelnasr et al. [21]. The implementation in [21] delegates the tallying computation to an off-chain untrusted administrator along with a dispute phase to catch the misbehavior of a malicious administrator. In contrast, we delegate all the computation to off-chain parties without loosing the dispute-free property. Regarding the gas cost, after carefully reviewing the Github code of [21], we found that their estimation does not take into account the '`verifyY`' step presented in the pseudocode of cast vote function in which the smart contract should verify that a voter `B` computes her blinding key Y_B correctly. We argue that this step is the major factor on the gas cost of this function. Moreover, they claimed that their design could be scaled theoretically to support 2^{256} voters because all transactions have constant gas cost except the two functions of registering voters and dispute phase which scales logarithmically with the number of voters since these functions verify the Merkle proof of membership. However, this claim also was based on ignoring the gas cost of the step '`verifyY`' which scales linearly with the number of voters.

6 Further Scalability Improvement

Recall that the gas consumption by the functions: `setVerifyingKey`(vk_{C_V}), `setVerifyingKey`(vk_{C_T}), CastVote, and SetTally, in the current design, increases linearly with the number of voters because the statement size of the zk-SNARK increases linearly with the number of voters. In this section, we present some modifications to the circuits so that the size of the statement becomes fixed, hence `setVerifyingKey`(vk_{C_V}) and `setVerifyingKey`(vk_{C_T}) consume fixed gas units. Regarding the gas consumption associated with invoking the other smart contract functions, we propose some modifications on Register, CastVote, and SetTally functions so that they also consume fixed gas units independent of the number of voters. As a result, the total gas paid by the administrator and each voter is constant, hence achieving very good scalability.

6.1 Fixed Statement Size

Let $commit_{PK} = H(pk_0^y || pk_1^y || \cdots || pk_{n-1}^y)$ denote the hash of the concatenation of the y-coordinates of the voters' public keys. Recall that in encryptedVoteGen circuit, the statement $\vec{s}$ =(V_B, i_B, $\{pk_i^y\}$) and the witness $\vec{w}$ =(v_B, x_B, $\{pk_i^x\}$). By moving $\{pk_i^y\}$ from $\vec{s}$ to $\vec{w}$ and adding $commit_{PK}$ to $\vec{s}$, we can construct a new circuit newEncryptedVoteGen with new $(\vec{s'}, \vec{w'})$ s.t.

$$\vec{s'} = (V_B, commit_{PK}, i_B), \qquad \vec{w'} = (v_B, x_B, \{pk_i^x\}, \{pk_i^y\})$$

The newEncryptedVoteGen circuit is encryptedVoteGen circuit in addition to a new constraint

$$\texttt{Assert}\ commit_{PK} = H(pk_0^y || pk_1^y || \cdots || pk_{n-1}^y)$$

Therefore, newEncryptedVoteGen has a fixed-size statement and independent of the number of voters, hence setting its verification key will consume fixed gas units. consequently, verifying the zk-SNARK proof will consume fixed gas units. In its turn, the smart contract should check the correctness of $commit_{PK}$ in CastVote function, since the public keys PK_i are known.

Similarly, let $commit_V = H(V_0^y || V_1^y || \cdots || V_{n-1}^y || D_0 || \cdots || D_l)$ denote the hash of the concatenation of the y-coordinates of the encrypted votes in addition to the integer numbers represented the sign-bits of the x-coordinates. Therefore, we can construct a new Tallying circuit with new $(\vec{s'}, \vec{w'})$ s.t. $\vec{s'} = (res, commit_V)$ and $\vec{w'} = (\{V_i^x\}$,$\{V_i^y\})$. In its turn, the smart contract should check the correctness of $commit_V$ in SetTallying function since the encrypted votes V_i are known.

As a result, the total gas units consumed by `setVerifyingKey`(vk_{C_V}) and `setVerifyingKey`(vk_{C_T}) become constant. In contrast, the gas consumption by CastVote and SetTallying functions still increase linearly with the number of voters and depend on how much the used hash function cost. Accordingly, this approach is efficient and can support a high number of voters only if the used hash function is cheap on the Ethereum, *e.g.,* SHA-256.

6.2 Fixed Gas Cost

Due to the structure of $commit_{PK}$ and $commit_V$, they must be calculated after registering all voters and casting all encrypted votes, respectively. Therefore, the voter who casts the first encrypted vote has to pay the linearly increasing gas cost of calculating $commit_{PK}$ on behalf of all the other voters. Similarly, the administrator has to pay the linearly increasing gas cost associated with calculating $commit_V$. To achieve a constant gas paid by the administrator and each voter, we restructure $commit_{PK}$ and $commit_V$ so that they can be calculated in a progressive manner. Hence their gas cost is distributed among all voters. Let the new $commit_{PK}$ be defined as follows:

$$commit_{PK} = H(H(\cdots H(H(0||pk_0^y)||pk_1^y)||\cdots)||pk_{n-1}^y)$$

Accordingly, the smart contract initializes $commit_{PK} := 0$ during its deployment, then during registering the public key PK_i, Register function updates $commit_{PK} := H(commit_{PK}||pk_i^y)$. At the end of the voters registration phase, the new $commit_{PK}$ value is ready to be used by CastVote function to verify the correctness of the encrypted vote sent by each voter.

Similarly, let the new $commit_V$ be defined as follows:

$$commit_V = H(H(\cdots H(H(0||V_0^x||V_0^y)||V_1^x||V_1^y)||\cdots)||V_{n-1}^x||V_{n-1}^y)$$

The smart contract initializes $commit_V := 0$, then during casting the encrypted vote V_i, CastVote function updates $commit_V := H(commit_V||V_i^x||V_i^y)$. By using both the two coordinates of V_i, there is no need of tracking the sign of the x-coordinates. At the end of the casting phase, the new $commit_V$ value is ready to be used by SetTallying function to verify the correctness of the tallying result sent by the administrator.

6.3 Performance Measurements

We evaluate the performance of the two modified designs in terms of the size of the common reference string (CRS) for encryptedVoteGen and SetTallying circuits (*i.e.,* the proving and verification keys sizes); their average proof generation time; and the gas consumption. associated with invoking the three functions: Register, CastVote, and SetTally. In particular, we evaluate our implementation using SHA-256 in Sec. 6.1 (referred as *SHA-256*), and SHA-256 and Poseidon [5] in Sec. 6.2 (referred as *progressive SHA-256* and *progressive Poseidon*). As depicted in Fig. 6, the CRS size and the average proof generation time for both circuits increase linearly with the number of voters in our original and modified designs. The highest value corresponds to the *progressive SHA-256* hashing case. As expected, SHA-256 cannot be presented by a friendly arithmetic circuit, *i.e.,* it generates a high number of constraints, hence a large proving key size and high zk-SNARK proof generation time. In contrast, Poseidon hash function is an arithmetic circuit friendly hash function. Regarding the gas cost, Register function consumes fixed gas units in all designs independent of the number of

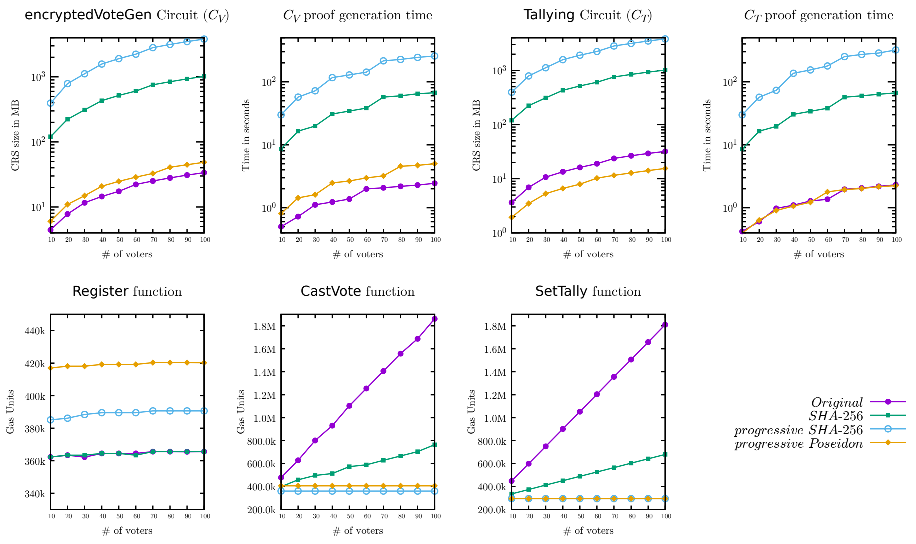


Fig. 6: Performance evaluation of our modified designs. '*Original*', '*SHA-256*', '*progressive SHA-256*', and '*progressive Poseidon*' refer to our original design, the modified designs in Sec. 6.1 using SHA-256, and Sec. 6.2 using the hash functions SHA-256, and Poseidon, respectively.

voters. However, it is the highest in *progressive Poseidon* hashing case. For both CastVote, and SetTally functions, their gas costs for *SHA-256* increase linearly with the number of voters, but *SHA-256* is still more scalable than our original design. In contrast, the two functions consume a fixed gas units in *progressive SHA-256* hashing and *progressive Poseidon* hashing as desired. However, the gas cost of *progressive Poseidon* hashing is slightly higher for CastVote function. The decision of which design to be used should be taken by the administrator and the voters since it presents a trade-off between the proof generation time and the money spent in the form of the gas fees.

7 Conclusion

We presented a dispute-free implementation for the Open Vote Network protocol as smart contracts in which all the heavy computations are performed off-chain. Then, we utilized zk-SNARKs to verify that all off-chain computations are performed correctly by their corresponding parties. Moreover, we developed a prototype of our design to assess its performance. Then, we enhanced its scalability by utilizing progressive hashing to achieve fixed zk-SNARK statement sizes and distribute the gas costs among all voters. As a result, the total gas paid by the administrator and each voter is constant, hence achieving very good scalability.

References

1. Adida, B.: Helios: Web-based Open-Audit Voting. In: USENIX security symposium. vol. 17, pp. 335–348 (2008)
2. Bellés-Muñoz, M., Whitehat, B., Baylina, J., Daza, V., Muñoz-Tapia, J.L.: Twisted Edwards Elliptic Curves for Zero-Knowledge Circuits. Mathematics **9**(23) (2021)
3. Buterin, V., Reitwiessner, C.: EIP-197: Precompiled contracts for optimal ate pairing check on the elliptic curve alt_bn128 (2017), https://eips.ethereum.org/EIPS/eip-197
4. Galal, H.S., ElSheikh, M., Youssef, A.M.: An Efficient Micropayment Channel on Ethereum. In: Data Privacy Management, Cryptocurrencies and Blockchain Technology – CBT 2019. LNCS, vol. 11737, pp. 211–218. Springer (2019)
5. Grassi, L., Khovratovich, D., Rechberger, C., Roy, A., Schofnegger, M.: Poseidon: A New Hash Function for Zero-Knowledge Proof Systems. In: 30th USENIX Security Symposium (USENIX Security 21). pp. 519–535 (2021)
6. Gritzalis, D.A.: Principles and requirements for a secure e-voting system. Computers & Security **21**(6), 539–556 (2002)
7. Groth, J.: Efficient Maximal Privacy in Boardroom Voting and Anonymous Broadcast. In: International Conference on Financial Cryptography – FC 2004. LNCS, vol. 3110, pp. 90–104. Springer (2004)
8. Groth, J.: Short Pairing-Based Non-interactive Zero-Knowledge Arguments. In: Advances in Cryptology - ASIACRYPT 2010. LNCS, vol. 6477, pp. 321–340. Springer (2010)
9. Groth, J.: On the Size of Pairing-Based Non-interactive Arguments. In: Advances in Cryptology – EUROCRYPT 2016. LNCS, vol. 9666, pp. 305–326. Springer (2016)
10. Hao, F., Ryan, P.Y., Zieliński, P.: Anonymous voting by two-round public discussion. IET Information Security **4**(2), 62–67 (2010)
11. Jivsov, A.: Compact representation of an elliptic curve point (Mar 2014), https://tools.ietf.org/id/draft-jivsov-ecc-compact-05.html
12. Kiayias, A., Yung, M.: Self-tallying Elections and Perfect Ballot Secrecy. In: Naccache, D., Paillier, P. (eds.) Public Key Cryptography – PKC 2002. LNCS, vol. 2274, pp. 141–158. Springer (2002)
13. Kshetri, N., Voas, J.: Blockchain-Enabled E-Voting. IEEE Software **35**(4), 95–99 (2018)
14. Li, H., Li, Y., Yu, Y., Wang, B., Chen, K.: A Blockchain-Based Traceable Self-Tallying E-Voting Protocol in AI Era. IEEE Transactions on Network Science and Engineering **8**(2), 1019–1032 (2021)
15. Li, Y., Susilo, W., Yang, G., Yu, Y., Liu, D., Du, X., Guizani, M.: A Blockchain-Based Self-Tallying Voting Protocol in Decentralized IoT. IEEE Transactions on Dependable and Secure Computing **19**(1), 119–130 (2022)
16. Maaten, E.: Towards remote e-voting: Estonian case. In: Prosser, A., Krimmer, R. (eds.) Electronic voting in Europe - Technology, law, politics and society, workshop of the ESF TED programme together with GI and OCG. pp. 83–90. Gesellschaft für Informatik e.V., Bonn (2004)
17. McCorry, P., Shahandashti, S.F., Hao, F.: A Smart Contract for Boardroom Voting with Maximum Voter Privacy. In: International Conference on Financial Cryptography and Data Security. LNCS, vol. 10322, pp. 357–375. Springer (2017)
18. Murtaza, M.H., Alizai, Z.A., Iqbal, Z.: Blockchain Based Anonymous Voting System Using zkSNARKs. In: 2019 International Conference on Applied and Engineering Mathematics (ICAEM). pp. 209–214. IEEE (2019)

19. Parno, B., Howell, J., Gentry, C., Raykova, M.: Pinocchio: Nearly practical verifiable computation. In: 2013 IEEE Symposium on Security and Privacy. pp. 238–252. IEEE (2013)
20. Sasson, E.B., Chiesa, A., Garman, C., Green, M., Miers, I., Tromer, E., Virza, M.: Zerocash: Decentralized Anonymous Payments from Bitcoin. In: 2014 IEEE Symposium on Security and Privacy. pp. 459–474. IEEE (2014)
21. Seifelnasr, M., Galal, H.S., Youssef, A.M.: Scalable Open-Vote Network on Ethereum. In: International Conference on Financial Cryptography and Data Security. LNCS, vol. 12063, pp. 436–450. Springer (2020)