Scalable Open-Vote Network on Ethereum

Mohamed Seifelnasr, Hisham S. Galal, and Amr M. Youssef

Concordia Institute for Information Systems Engineering,
Concordia University, Montréal, Québec, Canada

Abstract McCorry *et al.* (Financial Cryptography 2017) presented the first implementation of a decentralized self-tallying voting protocol on Ethereum. However, their implementation did not scale beyond 40 voters since all the computations were performed on the smart contract. In this paper, we tackle this problem by delegating the bulk computations to an off-chain untrusted administrator in a verifiable manner. Specifically, the administrator tallies the votes off-chain and publishes a Merkle tree that encodes the tallying computation trace. Then, the administrator submits the Merkle tree root and the tally result to the smart contract. Subsequently, the smart contract transits to an intermediate phase where at least a single honest voter can contend the administrator's claimed result if it was not computed correctly. Then, in the worst case, the smart contract verifies the dispute at the cost of an elliptic curve point addition and scalar multiplication, and two Merkle proofs of membership which are logarithmic in the number of voters. This allows our protocol to achieve higher scalability without sacrificing the public verifiability or voters' privacy. To assess our protocol, we implemented an open-source prototype on Ethereum and carried out multiple experiments for different numbers of voters. The results of our implementation confirm the scalability and efficiency of our proposed solution which does not exceed the current block gas limit for any practical number of voters.

Keywords: Open Vote Network, Merkle Tree, Smart Contract

1 Introduction

A blockchain is a decentralized append-only immutable ledger over a peer-to-peer network. It utilizes a consensus algorithm that ensures different users have access to a consistent ledger state. Furthermore, mining nodes have an economic incentive to behave honestly and compete in solving a cryptographic puzzle, referred to as Proof of Work (PoW), to receive block rewards.

As of November 2019, Ethereum capitalization exceeds 16 billion USD, which makes it the second most valuable blockchain after Bitcoin [2]. Ethereum is considered as a platform for running smart contracts in a world computer referred to as Ethereum Virtual Machine (EVM). Once a smart contract is deployed on the EVM, it becomes immutable, i.e., its code cannot be changed or patched afterward. Furthermore, it stays dormant until triggered either by a transaction

submitted from an Externally Owned Account (EOA) (i.e., a user account) or by a call from another contract. The underlying consensus protocol ensures that the smart contract state gets modified only as its code dictates.

In all transactions, the sender has to pay upfront in Ether for the execution of the contract's code. The computational complexity of a transaction is measured in gas, which can be bought for a *gas price* specified by the sender. Therefore, the transaction fee is the gas cost multiplied by the gas price. Furthermore, the sender also has to specify a *gas limit* which does not allow the transaction to burn more gas than the specified limit. During execution, if a transaction runs out of gas, then all the state changes are reverted while the transaction fee is paid to the miner. On the other hand, if the transaction is successful, then the sender gets the remaining gas.

Additionally, there exists a block gas limit, which limits the computational complexity of transactions in one block. Currently, the block gas limit is about 10,000,000 gas [1]. Obviously, it is important to minimize the gas cost of transactions in order to spend as little as possible on transaction fees. Furthermore, small gas costs are also crucial from a scalability point of view, since the less gas burnt for each transaction, the more transaction can fit into a single block.

McCorry *et al.* [10] presented the first implementation of the Open Vote Network protocol on the Ethereum blockchain. To hide their votes, voters send encrypted votes to the smart contract. These encrypted votes are accompanied by one-out-of-two Zero Knowledge Proof (ZKP) of either a 0 or 1 to prove the validity of the vote. Although their implementation tackles the voter privacy on Ethereum, it barely scaled up to 40 voters before exceeding the block gas limit. We identified two main reasons for this scalability problem from computation and storage perspectives. First, the smart contract computes the tally which involves running elliptic curve operations. Furthermore, this computation scales linearly with the number of voters. Secondly, at the deployment phase, the administrator sends the list of the eligible voters to be stored on the smart contract which also scales linearly with the number of voters.

Contribution. In this paper, we propose a protocol that efficiently reduces the computation and storage cost of the Open Vote Network without sacrificing its inherent security properties. More precisely, we make the following modifications:

1. We utilize a Merkle tree to accumulate the list of eligible voters. Thus, the smart contract stores only the tree root rather than the full list. Certainly, each voter will have to provide a proof-of-membership along with their votes.
2. We delegate the tally computation to an untrusted administrator in a verifiable manner even in the presence of a malicious majority. In fact, we require only a single participant, which could be a regulator or one of the voters, to be honest in order to maintain the protocol's security.

The rest of this paper is organized as follows. Section 2 presents a very brief review of some related work on voting protocols implemented on the Ethereum

blockchain. Section 3 presents the cryptographic primitives utilized in our protocol. Section 4 provides the design of the election contract and its execution phases. Also, it provides an analysis of the gas used in every transaction by the voter/election administrator. Lastly, Section 5 presents our conclusions.

2 Related Work

A cryptographic voting system is one that provides proof to each voter that her vote was included in the final tally. Public verifiability requires that the tallying process can be validated by anyone who wants to do so, even those who did not vote. Cryptographic voting systems should not leak any information about how the voter voted, beyond what can be inferred from the tally alone, including the cases where voters may deliberately craft their ballot to leak how they voted. Based on his mix network protocol [5], Chaum proposed the first cryptographic voting system in 1981. Interestingly, blind signature schemes [4], which formed the basis for the first cryptographic payment systems, have also been applied extensively in the design of e-voting protocols.

Traditionally, e-voting protocols rely on a trusted authority for collecting the encrypted votes from the voters to maintain the voters' privacy. Later, that trusted authority computes the final tally from the casted votes. The problem in this approach is giving a single centralized authority the full control of collecting and computing the tally. Instead, multiple authorities can be utilized for collecting the votes and in the tally computation phase, e.g., see Helios [3]. Yet, the collusion of the tally authorities is still a threat against voters' privacy. Removing the tally authorities completely was first accomplished by Kiayias and Yung [9] who introduced a boardroom self-tallying protocol. In a self-tallying voting protocol, once the vote casting phase is over, any voter or a third-party can perform the tally computation. Self-tallying protocols are regarded as the max-privacy voting protocols since breaching the voter privacy requires full collusion of all the other voters.

McCorry et al. [10] implemented the Open Vote Network protocol to build the first Boardroom voting on Ethereum. The protocol does not require a trusted party to compute the tally, however, it is a self-tallying protocol. Furthermore, each voter is in control of her vote's privacy such that it can only be breached by full collusion involving all other voters. To ensure the correct execution of votes tallying, the authors developed a smart contract that computes the votes tallying. Certainly, the consensus mechanism of Ethereum secures the tallying computation, however, running elliptic curve operations in smart contracts are cost-prohibitive. Therefore, the smart contract can tally a relatively small number of votes, up to 40, before consuming the block gas limit. Furthermore, a second drawback with this implementation is that at the deployment phase, the smart contract stores the list of all eligible voters. Technically speaking, storing large data on smart contracts is prohibitively expensive as the op-code `SSTORE` costs 20000 gas to store non-zero 32 bytes. For instance, storing a list of 500 voters' addresses will exceed the current block gas limit ($\approx$ 10 million gas).

3 Preliminaries

In this section, we briefly review the cryptographic primitives utilized in our protocol.

3.1 Merkle Tree

Merkle trees [11] are cryptographic accumulators with efficient proofs of set membership. Generally speaking, to accumulate a set of elements, one builds a binary tree where the leaf nodes correspond to the hash values of the elements. The *parent* nodes are assigned the hash of their children using a collision-resistant hash function. The set membership proof, known as *Merkle proof*, has a logarithmic size in terms of the number of leaves. For example, given a Merkle tree MT with a root r, to prove that an element $x \in MT$, the prover sends to the verifier a Merkle proof π which consists of the sibling nodes on the path from x to r as illustrated in Fig. 1. The verifier initially computes $r' \leftarrow H(x)$. Then, she iterates sequentially over each hash in π and reconstructs the parent r'. Finally, the verifier accepts the proof π if $r' = r$.

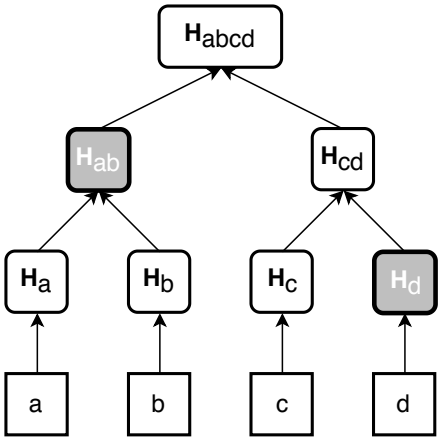

Figure 1: An example illustrating the Merkle proof for element $c \in MT$ which consists of the nodes H_d and H_{ab}

3.2 Pedersen Commitment

A commitment scheme enables a sender to commit to a value. After a while, the sender opens the commitment and reveals the committed value. The receiver

can check that the revealed value is the original committed value by doing the commitment again and checking that the two commitments match. Any commitment scheme must satisfy two properties: the hiding property and the binding property. The hiding property protects the sender's message from being compromised by the receiver while the binding property protects the receiver against a malicious sender who may change her committed message.

Pedersen commitment scheme [12] is computationally binding, perfectly hiding, and additively homomorphic. The commitment is computed as $c = g^m h^a$, where m is the message, a is a blinding value, and g and h are two generators. For the sender to reveal her commitment, she reveals both the message m and the blinding value a. Then, the receiver checks whether these values commit to the previously received commitment c.

3.3 Schnorr Zero-Knowledge Proof of Discrete Log Knowledge

A Zero-Knowledge Proof of Knowledge is an interactive protocol that runs between a prover and a verifier. It enables the prover to convince the verifier of her knowledge of a secret without revealing that secret to the verifier. Schnorr protocol [13] is a Σ protocol that consists of three interactions between the prover and verifier. These interactions are: commit, challenge and response. Let $v = g^s \ mod \ P$ where $s \in \mathbb{Z}_p$. In Schnorr ZKP, the prover knows a secret s (the discrete log of v) and she wants to convince the verifier of her knowledge without telling him the secret. ZKP protocol must achieve three properties: completeness, soundness, and zero-knowledge.

The Schnorr ZKP proceeds as follows: it starts by a commit phase where the prover sends the verifier her commitment $x = g^r \ mod \ p$ where $r \in \mathbb{Z}_p$. Then, the verifer sends back her challenge e where $e \in \mathbb{Z}_p$. Then, the prover responds with $y = (r - se) \ mod \ p$. In the end, the verifier checks $x = g^y \cdot v^e$.

3.4 Σ Protocol for a Commitment is either 0 or 1

Groth and Kohlweiss [8] presented a Σ protocol for commitment to either a 0 or 1, see Fig. 2. More precisely, the protocol proves the relation:

$$R = \{(c, (m, r)) \mid c = g^m \cdot h^r \ \texttt{where} \ m \ \in \{0, 1\} \ \texttt{and} \ r \ \in \ \mathbb{Z}_q\}$$

Using the homomorphic property of Pedersen commitments, $c^{x-f} c_b$ is a commitment to the message $m(x - f) + am \ = \ m(x - mx - a) \ = \ mx(1 - m) - am + am \ = \ x(1 - m)m$ which is 0 if $m \in \{0, 1\}$.

3.5 Open Vote Network

The Open Vote Network is a two-round self-tallying protocol that does not require a trusted party. In the first round, the administrator generates a cyclic group $\mathbb{G}$ of prime order q and a generator g. Then, each voter picks a random value $x_i \in \mathbb{Z}_q$ as a secret key and publishes her voting keys as g^{x_i} along with

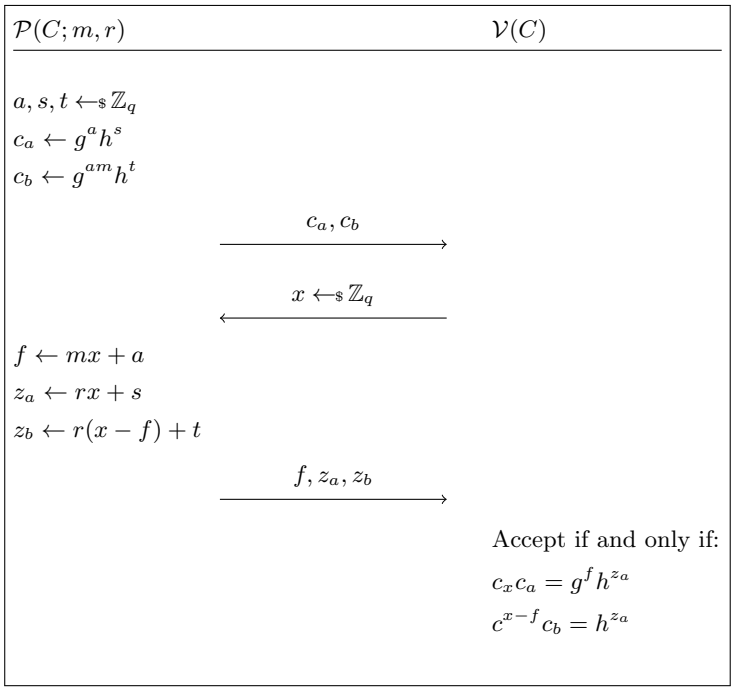

Figure 2: Σ protocol to prove that a committed m is either 0 or 1

a Schnorr proof of knowledge of discrete log. In the second round, each voter computes her blinding key as

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

By implicitly setting $Y_i = g^{y_i}$, then it is clear that $\prod_i Y_i^{x_i} = g^{\sum_i x_i y_i} = g^0 = 1$. Furthermore, note that the discrete log of $y_i = log_g(Y_i)$ is unknown to anyone. Therefore, one can utilize Y_i and g as the generators for Pedersen commitments. Hence, each voter broadcast commitments to their vote as $c_i = g^{v_i} Y_i^{x_i}$ along with a zero-knowledge proof that c_i is a commitment to either $v_i = 1$ or $v_i = 0$. As Pedersen commitment is additively homomorphic, then

$$\prod_i c_i = \prod_i g^{x_i y_i} g^{v_i} = g^{\sum_i x_i y_i + v_i} = g^{\sum_i v_i}$$

Finally, the tally result $\sum_i v_i$ can be easily obtained by performing an exhaustive search on the discrete log which is bounded by the number of voters.

4 Protocol Design and Implementation

In this section, we present the design of our proposed protocol and explain the various details regarding its implementation.

4.1 Protocol Overview

To bring scalability and efficiency to the deployment of Open Vote Network on Ethereum, we have to solve the computational and storage problems that we identified in [10]. First, we delegate the votes tallying process to an off-chain [6] untrusted administrator in a verifiable and efficient way. The proof verification of the delegated tally computation is logarithmic in the number of voters and involves a single elliptic curve point addition. Secondly, to significantly reduce the storage requirements of the smart contract deployment, we accumulate the list of eligible voters in a Merkle tree and store its root, which corresponds to a 256-bit hash value.

Our voting protocol is divided into six chronologically ordered phases. Starting with the deployment phase, the administrator Alice constructs a Merkle tree of all eligible voters $MT_{\mathcal{E}}$ and generates a set of public parameters. Then, she deploys the smart contract and initializes it with the $root_{\mathcal{E}} = root(MT_{\mathcal{E}})$ and a set of public parameters. Afterward, in the registration phase, all voters have to register their voting keys within its time window. For instance, suppose that Bob, who is one of the eligible voters, wants to cast his vote. Bob generates a voting key g^x along with Schnorr proof of discrete log knowledge π_x. Then, he submits g^x, π_x, in addition to a Merkle proof of membership π_{Bob}. Next, in the vote casting phase, the voters cast Pedersen commitments of their votes to the smart contract. Hence, Bob computes a commitment $c = g^v Y^x$ and generates a proof π_v that the committed vote v is either 0 or 1.

In the votes tallying phase, Alice obtains the committed votes stored on the smart contract, tallies them, and brute-forces the discrete log $\sum_i v_i$, which is bounded by the number of registered voters. We observe that the tally computation can be represented as a program that loops over the committed votes and accumulates their multiplications at each iteration. As a result, Alice can efficiently encode her the program execution trace by building a Merkle tree $MT_{\mathcal{C}}$ over the intermediate accumulated multiplication result of each iteration. Subsequently, she publishes $MT_{\mathcal{C}}$, for example, on the Interplanetary file system (IPFS) for public verifiability. Finally, she submits the $root_{\mathcal{C}} = root(MT_{\mathcal{C}})$ in addition to $\sum_i v_i$ to the smart contract.

Once $MT_{\mathcal{C}}$ is published, any voter or regulatory body can verify the tally computation trace done by Alice to determine whether the result has been computed correctly. One needs to count for scenarios where Alice could maliciously alter the inputs in one of the trace steps to affect the final tally result. Consequently, Bob, as an honest voter, can verify her computation trace and dispute her on the first invalid step i he finds. In other words, Bob does not have to verify the whole computation trace, instead, he simply disputes the first erroneous step. When the smart contract transits to the dispute phase, Bob submits

Merkle proofs for the inputs at step i encoded by Alice in $MT_{\mathcal{C}}$. After verifying the Merkle proofs, the smart contract will recompute the step i using the committed votes in its storage to detect whether Alice acted maliciously. If so, the smart contract will penalize her and reward Bob. On the other hand, if Bob tries to dispute a correct operation, the smart contract will simply reject Bob's transaction. Therefore, it is irrational for Bob to pay gas in that case. Eventually, in the reclaim phase, honest parties can request the release of their collateral deposits. In what follows, we explain the different phases of our protocol in more detail.

4.2 Phase 1: Smart Contract Deployment

In the beginning, Alice sets the interval for the phases: voters registration, vote casting, tally computation, dispute, and fund reclaim. She also establishes a list of all eligible voters. Then, she constructs a Merkle tree $MT_{\mathcal{E}}$ of the voters in this list. Then, Alice publishes it so that each voter can construct her own Merkle proof of membership. Upon deploying the contract, Alice sends the interval of each phase and the $root_{\mathcal{E}} = Root(MT_{\mathcal{E}})$ to the contract rather than storing the full list in the smart contract permanent storage.

Initialize: upon receiving $(root_{\mathcal{E}}, T_1, T_2, T_3, T_4, T_5)$ from administrator `A`:
`Assert` $value = F$
`Store` $root_{\mathcal{E}}, T_1, T_2, T_3, T_4, T_5$
`Init` $voters := \{\}$, $votes := \{\}$, $keys := \{\}$ $index := 1$

Figure 3: Pseudocode for deployment of the smart contract.

As illustrated in Fig. 3, the voting administrator deploys the voting contract on Ethereum with the following set of parameters:

1. $root_{\mathcal{E}}$: Root of the Merkle tree of the eligible voters.
2. T_1, T_2, T_3, T_4, T_5: The block heights which define the end of the phases: registration, vote casting, tally computation, dispute, and reclaim, respectively.
3. F: A collateral deposit that is paid by Alice and the voters. This deposit is used to penalize malicious behavior if any.

4.3 Phase 2: Voters Registration

This phase starts immediately after the contract deployment where interested voters can participate by registering their voting keys. For instance, Bob as one of the eligible voters generates a voting key g^x along with Schnorr proof of DL π_x. Then, he submits a transaction containing g^x, π_x, a Merkle proof of membership

RegisterVoter: upon receiving (g^x, π_x, π_B) from voter `B`:

```
Assert value = F
Assert T < T1
Assert verifyMerkleProof( π_B, B, root_E)
Assert verifyDL( g^x, π_x)
Set keys[index] := g^x
Set voters[index] := B
Set index := index + 1
```

Figure 4: Pseudocode for register voter function

π_{Bob} as parameters, and pays a collateral deposit F as shown in Fig. 4. The smart contract ensures that registration transactions are accepted only within the allowed interval and verifies both the Schnorr proof of DL knowledge and the Merkle proof of membership. For verifying membership of voters in the $MT_{\mathcal{E}}$, we use the *VerifyMerkleProof* algorithm implemented in [7]. Furthermore, recall that in the Open Vote Network, voters have fixed positions which allow them to properly compute Y_i. In our protocol, we impose that each voter takes the order at which his voting keys were stored in the smart contract (i.e., an index in the array of voting keys).

4.4 Phase 3: Vote Casting

After all the voting keys have been submitted, voters can generate Pedersen commitments to their votes. More precisely, suppose Bob's voting key is stored at index i, then he computes:

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

Furthermore, since $y_i = log_g(Y_i)$ can only be known by the full collusion among all voters, which is assumed not to be the case in the Open Vote Network protocol, then Bob can safely use g and Y_i as the generators for Pedersen commitment. Hence, Bob commits to his vote v as $c = g^v Y_i^{x_i}$ where the blinding value is his secret voting key x_i. Next, Bob submits a transaction containing c, Y_i, and a zero-knowledge proof π_v that the committed v is either 0 or 1. The smart contract will store the commitment c if the transaction is sent within the right time window and the proof π_v is verified successfully as shown in Fig. 5.

4.5 Phase 4: Tally Computation

This is the phase in our implementation which aims to bring scalability to the Open Vote Network protocol. Basically, we show how to significantly reduce

CastVote: upon receiving $(c,\ Y,\ \pi_v)$ from voter `B`

`Assert` T_1 `<` T `<` T_2

`Assert verifyZeroOrOne(`$c,\ Y,\ \pi_v$`)`

`Set` $index :=$ `IndexOf(`$B, voters$`)`

`Set` $votes[index] := c$

Figure 5: Pseudocode for cast vote function

the transaction fees by delegating the tally computation to an untrusted administrator, Alice, in a publically verifiable manner. Suppose that the vector $\mathbf{c} = (c_1, ..., c_n)$ contains the n committed votes sent to the smart contract. We observe that the tally computation $\prod_i c_i = \prod_i g^{v_i} Y_i^{x_i}$ can be computed by a program that iterates over the vector $\mathbf{c}$ and accumulates intermediate multiplication result as shown in Fig. 6.

```
def TallyVotes(c: array[]):
    t = 1
    for i=1 to n:
        t = Mul(c[i], t)
    return t
```

Figure 6: Program tally function

The program execution trace is represented as a $4 \times n$ array where the first column denotes the step number, and the remaining columns denote the two input operands and the accumulated multiplication result as shown in Table 1.

Table 1: Computation tally execution trace

Step i	c_i	t_{i-1}	t_i
1	c_1	$t_0 = 1$	$t_1 = c_1$
2	c_2	t_1	$t_2 = c_2 \cdot t_1$
.	.	.	...
n	c_n	t_{n-1}	$t_n = c_n \cdot t_{n-1}$

Afterwards, Alice constructs a Merkle tree $MT_{\mathcal{C}}$ to encode the result t_i at each row. Specifically, the data for each leaf node is formatted as $(i||t_i)$ where $||$ denotes concatenation. Furthermore, she brute-forces $log_g(t_n) = \sum_i v_i$ which corresponds to the sum of the committed votes. Finally, she creates a transaction

SetTally: upon receiving $(res, root_{\mathcal{C}})$ from administrator `A`:
`Assert sender = A`
`Assert` $T_2 <$ `T` $< T_3$
`Store` res, $root_{\mathcal{C}}$
`Set` $tallySubmitted := true$

Figure 7: Pseudocode for set tally function

to the smart contract with the parameters $root_{\mathcal{C}} = root(MT_{\mathcal{C}})$ and the tally result $res = \sum_i v_i$ as shown in Fig. 7. The smart contract stores these parameters provided that the transaction within the interval of this phase.

4.6 Phase 5: Tally Dispute

After publishing the Merkle tree $MT_{\mathcal{C}}$ on IPFS, any voter or regulatory body can verify the correctness of the intermediate accumulated multiplication result of each trace step. Alice could attempt to maliciously affect the tally result by using a different vote commitment c_i' which is different from the c_i stored on the smart contract. For example, suppose Alice incorrectly set $t_i = c_i' \cdot t_{i-1}$. Note that, she could make multiple errors, however, it is sufficient to dispute the first one. Bob disputes her by sending i, t_i, t_{i-1} along with Merkle proofs π_i, π_{i-1} to the smart contract as shown in Fig. 8.

There are three different cases for how the smart contract handles the dispute based on the parameter i:

1. When the disputed step is the first one (i.e., $i = 1$), then the smart contract will only verify whether $t_1 \neq c_1$ since we assume $t_0 = 1$.
2. For other steps where $i \in [2, n]$, the smart contract will verify the Merkle proofs π_{i-1} and checks if $t_i \neq c_i \cdot t_{i-1}$.
3. Finally, the last step is related to the case where Alice has encoded the correct computation trace. However, she submitted an incorrect discrete log res in the previous phase. Thus, the smart contract will test whether $g^{res} \neq t_n$.

If any of these cases is verified successfully, the smart contract will reward Bob and set the flag *disputed* to prevent Alice from reclaiming her collateral deposit in the reclaim phase.

Dispute: upon receiving $(i,\ t_i,\ t_{i-1},\ \pi_i,\ \pi_{i-1})$ from voter `B`:

```
Assert T_3 < T < T_4
Assert disputed ≠ true
Assert VerifyMerkleProof(π_i, (i||t_i), root_C)
Set c_i := votes[i]
Set n := votes.length
IF (i > 1 and i ≤ n)
      Assert VerifyMerkleProof(π_i, (i − 1||t_{i−1}), root_C)
      IF t_i ≠ c_i · t_{i−1}
            Set disputed := true
IF (i = 1 and t_i ≠ c_i)
      Set disputed := true
IF (i = n and g^res ≠ t_i)
      Set disputed := true
IF disputed := true
      B.transfer(F)
```

Figure 8: Pseudocode for the dispute function

4.7 Phase 6: Reclaim

After the dispute phase, each honest participant can submit a transaction to reclaim her collateral deposit. The smart contract checks whether the sender has not been refunded before. Then, it checks whether the sender has behaved honestly in following the specified protocol steps. More precisely, if the sender is one of the voters, then the smart contract checks if that voter has already submitted the vote commitments. On the other hand, if the sender is the administrator, then it checks whether the flag disputed is not set. On success, the smart contract sends the deposit back to the sender as shown in Fig. 9.

Reclaim: upon receiving() from a sender:

```
Assert T_4 < T < T_5
Assert refund[sender] = false
Assert (sender ∈ voters and votes[sender] ≠ null) or
(sender = A and tallySubmitted and disputed = false)
Set refund[sender] := true
sender.transfer(F)
```

Figure 9: Pseudocode for reclaiming collateral deposit

4.8 Gas Cost Analysis

In order to assess our protocol, we developed a prototype and tested it with a local private Ethereum blockchain. The prototype is available as open-source on the Github repository[1]. On the day of carrying out our experiments, during November 2019, the ether exchange rate to USD is 1 ether $\approx$ 140$ and the gas price is approximately 10 $Gwei = 10 \times 10^{-9}$ ether. The genesis initialization file of the local blockchain contains {"$byzantiumBlock$" : 0} attribute in order to support our elliptic curve point addition and scalar multiplication over *alt_bn128* curve [14]. The test scenario is implemented with 40 local Ethereum accounts to compare our results with the implementation of McCorry *et al.* [10]. In Table 2, we show the gas used per voter/administrator for every function in the smart contract and the corresponding gas cost in USD.

Table 2: The gas cost for functions in the voting contract

Function	Gas units	Gas cost (USD)
CryptoCon	883,113	1.24
VoteCon	1,858,544	2.06
RegisterVoter	206,433	0.28
CastVote	346,655	0.49
SetTallyResult	64,723	0.09
Dispute	98,310	0.14
Reclaim	50,104	0.07

It should be noted that, in our implementation, the total gas paid by the administrator is constant. In particular, the administrator pays the gas for the deployment of two smart contracts: CryptoCon and VoteCon, in addition to a transaction `setTallyResult`. Neither any of these transactions involve operations that depend on the number of voters. On the other hand, for the voters, the transaction cost of `RegisterVote` scales logarithmically with the number of voters since it verifies the Merkle proof of membership. Similarly, the transaction `Dispute` scales logarithmically as it verifies two Merkle proofs in addition to carrying two elliptic curve operations (one point addition and one scalar multiplication) at maximum. All the other transactions have a constant cost.

Although the Open Vote Network protocol is suitable for a small number of voters, we carried out some experiments to determine the highest number of voters that can be supported in our prototype without exceeding the block gas limit. Recall that all transactions have constant gas cost except `RegisterVoter` and `Dispute` which scales logarithmically with the number of voters due to verification of Merkle proofs. Furthermore, the primitive unit of storage on Ethereum is `uint256`, hence theoretically the largest number of voters supported by the smart contract is 2^{256}. Therefore, in the `RegisterVoter` transaction, the voter

[1] https://github.com/HSG88/eVoting

sends a Merkle proof of membership which consists of 256 hash values (i.e., 256×32 bytes). Interestingly, we found the total gas cost in this theoretical case to be 667,254 $\approx$ 6.6% of the current block gas limit. Furthermore, we followed the same approach to find the gas cost for the `Dispute` transaction. In that case, the smart contract verifies two Merkle proofs and carries out elliptic curve single scalar multiplication and point addition at a total estimated gas cost 1,426,593 $\approx$ 14.3% of the current block gas limit. Since these two numbers serve as upper bounds for the gas cost in any practical scenario, the results of this experiment clearly confirm that the operations within the smart contract in our prototype does not limit the number of supported voters in practice.

In McCorry *et al.* implementation, all computations are performed on the smart contract. Thus, while there is no dispute phase, the number of voters it can support is significantly limited. For the administrator, the gas used comes from VoteCon, CryptoCon, Eligible, Begin Signup, Begin Election and Tally transactions [10] which is equal to about 12 million gas units. For the voter, the gas cost comes from Register, Commit and Vote transactions which sum to 3 million gas units. Table 3 compares the total gas cost in our implementation versus theirs for the same number of the 40 voters.

Table 3: Gas cost comparison between the two implementations

Sender	Our Implementation	McCorry *et al.* [10]
Voter	701502	3323642
Admin	2856484	12436190

5 Conclusion

In this paper, we presented a protocol that efficiently reduces the computation and storage cost of the Open Vote Network without sacrificing its inherent security properties. More precisely, we utilize a Merkle tree to accumulate the list of eligible voters. Additionally, we delegate the tally computation to an untrusted administrator in a verifiable manner even in the presence of a malicious majority. In fact, we require only a single participant, which could be a regulatory body or one of the voters, to be honest in order to maintain the protocol's security. Also, we developed a prototype to assess our protocol and carried out multiple experiments. The results of our experiments confirm that our prototype is efficient and can support a very large number of voters without exceeding the current block gas limit.

References

1. Ethereum gaslimit history (2018). `https://etherscan.io/chart/gaslimit`. [Online; accessed 24-Novemebr-2019].
2. Top 100 Cryptocurrencies by Market Capitalization. `https://coinmarketcap.com`. [Online; accessed 22-Novemebr-2019].
3. B. Adida. Helios: Web-based open-audit voting. In *USENIX security symposium*, volume 17, pages 335–348, 2008.
4. D. Chaum. Blind signatures for untraceable payments. In *Advances in cryptology*, pages 199–203. Springer, 1983.
5. D. Chaum. Untraceable electronic mail, return addresses and digital pseudonyms. In *Secure electronic voting*, pages 211–219. Springer, 2003.
6. J. Eberhardt and S. Tai. On or off the blockchain? insights on off-chaining computation and data. In *European Conference on Service-Oriented and Cloud Computing*, pages 3–15. Springer, 2017.
7. H. S. Galal, M. ElSheikh, and A. M. Youssef. An efficient micropayment channel on ethereum. In *Data Privacy Management, Cryptocurrencies and Blockchain Technology*, pages 211–218. Springer, 2019.
8. J. Groth and M. Kohlweiss. One-out-of-many proofs: Or how to leak a secret and spend a coin. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 253–280. Springer, 2015.
9. A. Kiayias and M. Yung. Self-tallying elections and perfect ballot secrecy. In *International Workshop on Public Key Cryptography*, pages 141–158. Springer, 2002.
10. P. McCorry, S. F. Shahandashti, and F. Hao. A smart contract for boardroom voting with maximum voter privacy. In *International Conference on Financial Cryptography and Data Security*, pages 357–375. Springer, 2017.
11. R. C. Merkle. Protocols for public key cryptosystems. In *1980 IEEE Symposium on Security and Privacy*, pages 122–122. IEEE, 1980.
12. T. Pedersen and B. Petersen. Explaining gradually increasing resource commitment to a foreign market. *International business review*, 7(5):483–501, 1998.
13. C.-P. Schnorr. Efficient signature generation by smart cards. *Journal of cryptology*, 4(3):161–174, 1991.
14. E. P. Team. Ethereum improvement proposals, 2017. `https://github.com/ethereum/EIPs`.