Decentralized, Transparent, Trustless Voting on the Ethereum Blockchain

Fernando Lobato Meeser
lobato.meeser.fernando@hotmail.com

November 21, 2017

Abstract

We present the first implementation of a voting system as a smart contract running on Ethereum that uses threshold keys and linkable ring signatures to provide a transparent and robust system that could be implemented for medium size elections. Each voter is in control and can monitor his vote while remain anonymous amongst a set of users. The protocol minimizes centralization by the use of threshold cryptography, allows for the voting to be tallied by anybody and does not require every single user to vote for tallying to be precise. All the execution of the protocol is ensured by the safety of the Ethereum protocol. We deployed the contract to the Ethereum test network and provide some analysis on feasibility and costs.

1 Introduction

Electronic voting has been a longtime interest topic for cryptography because of its need to guarantee transparency and immutability while maintaining voter privacy. Various voting schemes have been proposed in the past using sophisticated cryptographic techniques to provide these needs. Chaum proposed the first concept of an electronic voting scheme[1]. Benaloh brought forward a verifiable election system that uses threshold encryption to ensure votes can't be tallied until the election is over and a vote shuffle with zero knowledge proofs to ensure the privacy of voter [2, 3]. The open vote protocol [4] presented the first self-tallying voting system using homomorphic encryption and was recently implemented as a smart contract over Ethereum [5] achieving complete decentralization but requiring all participants to cast a vote for the self-tallying function to work.

The advent of Blockchain brought a new area of research to life on how to create trustless and decentralized systems. The first Blockchain, Bitcoin [6] is an append-only ledger of transactions that does not require any central institution to settle the validity of transactions. This sparked the invention of Ethereum [7] a blockchain with a Turing complete programming language that allows users to create guaranteed to execute computer programs called smart contracts.

Voting systems have a tradeoff in decentralization between the anonymity of the participants and the robustness of the system. This paper focuses on how to run an election process in which all payers can have cryptographic assurance that the outcome is valid in an electronic untrusted network. We begin by using the blockchain as a bulletin board and a global computer to publish all votes and provide users with an authenticated public system. The scheme uses threshold encryption to ensure voters can't tally election results before the election is over and linkable ring signatures [8] to protect voter privacy among a set of voters without allowing a participant to vote twice or to change his vote if the system requirements demand it.

Threshold encryption can distribute trust between various third parties by splitting a secret amongst n parties and requiring only a subset t of n to reconstruct the secret. While this approach gives a degree of centralization to the third parties holding these secret splits, we may assume a minimum level of trust on them since all them would need to be corrupted to exploit the system. It's worth pointing out that even if t out of n third parties were to be corrupted or compromised and release their keys during a voting phase, this would not compromise the consistency of the votes of each participant in any way.

On the other hand, linkable ring signatures allow a participant to present a valid signature from a set of keys without revealing which key generated the signature. It can be seen as a zero proof of knowledge of a key inside a set of keys. This allows a participant to remain anonymous among a set of participants. The larger the number of users involved in the signature the more anonymous it becomes. This presents a tradeoff in the current scheme. Public blockchains are currently experimenting scaling difficulties [9, 10] because of the redundancy needed to ensure decentralization. Every node in the network must run the complete state of the system. In the case of Ethereum this means every node must run all smart contract code. Linkable signatures verification time is linear to the number of users. Therefore, we must find a number that allows for voter privacy but does not put a large burden in the system.

Contribution
We provide the implementation of a voting scheme as an Ethereum smart contract that uses threshold keys and linkable ring signatures to provide a transparent and robust system that can be implemented for medium size elections like county or municipal type. Alongside we have two python programs, one for threshold keys that allows an election authority to generate and distribute a secret and one for linkable ring signatures that allows any user to create a valid ring signature. For the remaining sections of this paper in Section 2 the security scheme and infrastructure is presented, Section 3 describes our scheme and implementation over Ethereum, Section 4 describes and presents our results. Finally, in section 5 the concluding remarks are provided.

2 Background

2.1 Ethereum

The Ethereum blockchain is an ordered transaction based state machine. It has a distributed P2P network of computers that redundantly process all transaction being broadcasted into the network. There are two types of accounts in Ethereum:

- **Externally owned accounts (EOA)**: This is a key-pair controlled by a user that allows him to send transactions into the network.
- **Contract account**: This is a contract deployed on the blockchain through a transaction by an EOA that is now controlled by its code.

Both accounts can hold any given amount of Ethereum native token called Ether. Ether is the currency inside Ethereum to pay for using the network's computational resources. A smart contract can't execute code by itself. It need interaction from EOA to execute functionality. Any EOA can send a transaction to the contract address invoking a function of its code. However, when invoked, a contract can call other contracts. To run this, the user needs to pay for 'gas'. Gas is a metric to standardize the cost of executing code on the network, each assembly operation has a fixed gas cost based on its execution time. When sending a transaction, the user specifies how much Ether he is willing to pay for each unit of gas. This creates a market amongst miner and users to include their transactions in the blockchain.

The main fields of an Ethereum transaction are:

- **From (v, r, s):** ECDSA signature for an EOA address to authorize transaction.
- **To:** Destination address (this can be EOA or contract address).
- **Value:** Amount of Ether being transferred (if any).
- **Data:** This is where the compiled code goes. If a user is generating a new contract, he would put the contract code here and send a transaction without a destination address. If a user is calling a function inside a contract he would put the function call in this section and send a transaction to the contract.
- **Gas Price:** How much Ether the user is willing to pay for each unit of gas the contract consumes.
- **Start Gas:** The maximum amount of gas the user is willing to pay of the transaction.

We can see the Ethereum blockchain as a state machine. Each block has a set of transactions that change the current state of the whole blockchain. Contract accounts have access to persistent storage; therefore, each function execution creates a new state. Sending an ether from an account to another requires the values to be updated in all the computers inside the network. Executing a function of the contract is the same. Currently there are around 20,000 Ethereum nodes running the protocol in the main network. Each transaction must be mined into a block using the Proof-of-Work scheme and then all nodes most update the state with the new block.

This gives us a decentralized computing environment. Any user can sync the blockchain and have a complete copy of the history of all transactions.

2.2 Threshold Encryption

For an election process to be robust and valid, no one should be able to tally the results before every single vote is casted. This presents a challenge since the blockchain is a completely public computing environment. Anybody can see all of the transactions being published. Therefore, we need users to publish their vote publicly but the vote must not be identifiable. On the OpenVote protocol, users submit their vote with a signature containing all other voter's public keys. When everybody submits their vote, one can tally the election result. But this requires everyone to submit their vote. To go around this we are using Threshold encryption.

Threshold encryption is a special type of public key encryption where n players generate a key pair. The private key is split into N different shares and each share is held by a player. To reconstruct the private key is required t out of n players to interactively work together and publish their secret share. Anybody can encrypt a message using the corresponding public key, but the message can only be decrypted until the private key is reconstructed.

There are many schemes based on RSA and the DLP (Discrete Logarithm Problem), nevertheless, currently Ethereum can only perform arithmetic operations at most over 256 unsigned integers, so we chose a for Elliptic Curves based on the Elliptic Curve Discrete Logarithm Problem for its higher level of security per bit.

The (t, n) threshold scheme is defined as follows:

All users must agree on $(F_q, E(F_q), G, l)$ where F_q is a finite cyclic group of prime order q. $E(F_q)$ is an elliptic curve over the finite group F_q, G is a base point on the curve $E(F_q)$ and l is the order of the base point.

The n players get together and generate a secret d and its corresponding public key $Q = d \cdot G$. To split the secret into n shares we must first define a polynomial $f(x)$ where f_j denotes the coefficient of the x^i variable inside the polynomial. Each f_j must be random value.

We therefore have a secret polynomial:

$$f(x) = (\sum_{i=0}^{t-1} f_i \cdot x^i) \pmod{l}$$

Before destroying the secret d, the value inside the f_0 coefficient must be set to d. Now each secret share S_i can be calculated as

$$S_i = f(i)\{i = 1, 2, ...n\}$$

.
Now we can turn the parameters that we used to generate the secret split into public keys so that anybody that holds a secret can verify their validity. We are going to generate a set of public parameters F.

$$F_i = f_i \cdot G\{i = 0, 1, ..., t-1\}$$

With this, any player can verify their secret share is valid by verifying the equation.

$$t_i \cdot G = \sum_{j=0}^{t-1} i^j \cdot F_j$$

t player can reconstruct the private key using Lagrange Polynomial Interpolation:

$$d = \sum_{j=0}^{t} t_j \cdot \prod_{h=1, h\neq j}^{t} \frac{h}{h-j} \pmod{l}$$

Any user can encrypt a message m using the public key Q which can only be decrypted once t out of n player cooperate. This can be done using standard elliptic curve encryption.

Encrypt

1. Select random $k \in Z_l$
2. $P = k \cdot G = (x_1, y_1)$, x_1 should not be 0.
3. $k \cdot Q = (x_2, y_2)$, x_2 should not be 0.
4. $c = m \cdot x_2$, send ciphertext (P, c)

Decrypt

Once the private key d has been reconstructed using the method above we can the decrypt the ciphertext (P, c) in the following way.

1. $(x_2, y_2) = d \cdot P$
2. $m = c \cdot x_2^{-1}$

2.3 Linkable Ring Signatures

Another very important requirement for an authentic electoral process is that each vote must be completely anonymous and only authenticated voters can be registered. Ethereum transactions are pseudo anonymous because anybody can see the corresponding public key from which a vote came from. An election authority that registers users into the election contract could have a record linking public keys to individual voters. This would jeopardize their anonymity since the election authority could match voters and votes after decrypting results.

Chaum and Heyst introduced group signatures [11], a scheme by which a set of users each with their corresponding key-pair called a group. In this group, we have a manager that interactively forms the group. Any member of the group can sign a message in behalf of the group and a receiver can verify the validity of the signature without revealing the individual signer. In the case it's required, an individual signer can be revealed by a group manager. To avoid the require trust setup of a group manager and produce scheme without privacy revocation, Rivest, Shamir and Tauman formalized ring signatures [12]. Ring signatures allow a member inside a group of users to sign a message on behalf of the group without their approval or assistance. The anonymity of this scheme can't be revoked. To ensure we can detect a user presenting multiple signatures in behalf of the group we can use a linkable ring signature, where each private key leaves a link that can't be tied back to a user but can be used to know if a user signed a message in multiple occasions[8].

The scheme being used takes the DLP and is provable under the random oracle model, using a cryptographic hash function as a random oracle. We adapted this scheme for use over elliptic curves by hashing into an elliptic curve using the 'Try-and-Increment' Method [13].
We once again have $(F_q, E(F_q), G, l)$ where F_q be a finite cyclic group of prime order q. $E(F_q)$ is an elliptic curve over the finite group F_q, G is a base point on the curve $E(F_q)$ and l is the order of the base point.

Let H_1 be a cryptographic hash function that takes an input and can map it to a number inside F_q. More formally, $H_1 : \{0,1\}^* \mapsto \mathbb{Z}_q$.

Let H_2 be a cryptographic hash function that can map an input to a point into an elliptic curve.

We asume n users that form our group where each user has their corresponding private key x_i and their public key $y_i = x_i \cdot G$. We call the list of all public keys $L = \{y_1, ..., y_n\}$.

Generate Signature

The user with the private key x_i want to sign message $m \in \{0,1\}^*$ with L.

1. Compute $H = H_2(L)$.
2. Compute $K = x_i \cdot H$.
3. Pick a random number $u \in \mathbb{Z}_q$ and compute
$$c_{i+1} = H_1(L, K, m, G \cdot u, H \cdot u).$$
4. For $j = i+1, .., n, 1, .., i-1$ Pick $s_j \in \mathbb{Z}_q$ and compute:
$$c_{j+1} = H_1(L, K, m, G \cdot s_j + y_j \cdot c_j, H \cdot s_j + K \cdot c_j).$$
5. Compute $s_i = u - x_i \cdot c_i \pmod{q}$.

The signature is $SIG(m) = (c_1, s_1, ..., s_n, K)$.

Validate Signature

Given a signature $SIG(m) = (c_1, s_1, ..., s_n, K)$ over a message m and a list of public keys L anybody can verify the signature.

1. Compute $h = H_2(L)$
2. For $i = 0, ..., n-1$ compute:
$$\alpha_i = G \cdot s_i + y_i \cdot c_i$$
$$\beta_i = H \cdot s_i + K \cdot c_i$$
$$c_{i+1} = H_1(L, K, m, \alpha_i, \beta_i)$$
3. Verify wether $c_1 = H1(L, K, m, \alpha_n, \beta_n)$. Accept or Reject based on this.

3 Voting Scheme

In our scheme, we assume an election authority that will be in charge of coordinating the third parties holding the secret shares, publish the election contract to the blockchain and registering users into the contract.

Before uploading the contract to the blockchain, n third parties and the election authority must get together in a public gathering, generate a key-pair. Split the key pair into n secrets where each third party will take a secret with them and destroy the original secret without keeping a copy.

Next the election authority must submit the contract with the election rules to the blockchain, publish the code and provide the address of the contract. This way any voter can compile the code and verify it's the same contract.

The election is broken into the following phases.

- Setup

  Once the contract is deployed it is automatically in Setup phase. At this time, the contract owner (election authority) must submit all the parameters for an election to take place. Only the owner of the contract can set this information.

  - Registration start and end time.
  - Voting start and end time.
  - Voting options
  - Number of secret shares for threshold key reconstruction (t)
  - Number of existing shares for threshold key reconstruction (n)
  - Public parameters to verify validity of secret share F
  - Threshold Key Q

  Once this parameters are published and validated, the contract moves to registration phase.

- Registration

  Any user can query the contract to see the registration period and go with the election authority to register their public key y_i for casting a vote. The election authority registers they key into the contract and could keep a second log of which user registered which keys for auditing purposes. This will not impact the anonymity of the user since he will generate a vote with a LRS (linkable ring signature) which takes a big number of public keys to hide in the crowd.

  Once the voters public key has been submitted to the blockchain, he can verify this and wait for voting phase.

- Voting

  Each voter can query the blockchain, obtain the threshold public key and encrypt their vote. Then they can query the set of all other registered users in the contract and send the encrypted vote signed with a LRS. They can use any Ethereum account to submit their vote to the contract and verify that it was casted successfully.

- Secret Reconstruction

  Once the voting phase is finished, we have all the encrypted votes recorded on the Ethereum blockchain. In this phase, all third parties holding a secret share can submit their secret share into the contract. When t out of n shares are in the contract, the secret key can be recalculated.

- Tallying

  Finally, the contract can decrypt the results of the election and anybody can tally re results and see the winner.

3.1 Implementation Over Ethereum

Our implementation contains an Ethereum smart contract for defining the election and a library for verifying ring signatures all written in solidity. The contract for the election defines the phases specified before. A python library for generating and verifying linkable ring signatures with a program to sign a vote. A python library for generating and reconstructing threshold keys. A variety of scripts and tools to create your own Ethereum private network and tests for the contract. All the code is open and publicly available[1].
Due to Ethereum primitive computing environment some things were adjusted. Since the verification time of a ring signature is lineal to the number of public keys used to generate the signature, users don't sign with all the set of public keys. They sign with a sub-set of size R. This constant R is defined by the election authority during setup time. The bigger R the more anonymous the scheme becomes, but the more inefficient.

We also present a small HTML/JavaScript web application to use the smart contract in GUI. The application is divided into three sections.

- Admin (admin.html)

  In this section, the contract owner can authenticate with Ethereum and set the parameters for an election. He can then also register eligible voters into the contract.

- Voter (voter.html)

  A voter user has a simple interphase where we can visualize all the information for the election contract. Once an election has a setup, he can look at dates, parameters and download the threshold key. Every time a voter is registered or a vote is casted into the contract it can be viewed. When a user wants to cast his vote, he can query all the public keys that are part of its sub-ring by submitting his public key. Finally, any voter can tally and view the results of the election locally once the secret key has been reconstructed.

- Secret Holder (secret.html)

  In this section, the third parties holding secret shares can submit their shares to the blockchain once all votes are casted. Since Ethereum has no current support for floating point arithmetic, the private key can't be reconstructed in the blockchain. But anybody can download all the secret shares to re-compute the key with the python script provided and re upload it to the

[1] https://github.com/fernandolobato/decentralized-blockchain-voting

blockchain.

To generate a threshold key and split it into multiple files we created a lightweight purely pythonic library for threshold cryptography using ECDSA pythons' library. It has a script by which anybody can generate a threshold key giving parameters t and n. Reconstruct a private key from a sub secret share of size t. Finally encrypt or decrypt a message using a threshold key.
For a voter to cast a vote, we generated a lightweight purely pythonic linkable ring signature library using ECDSA pythons' library. the library has functions for signature generation and verification. We generated a script for the user to vote that can take a set of public keys, a private key corresponding to one of the public keys, a threshold key and a message (which represents the vote). The program will encrypt the message using the threshold key and generate a ring signature with the private key provided over the set of public keys. It outputs in a format that is consumable by our web application. At the same time, all the information that can be queried from the web application (set of public keys and threshold key) is given in a format that is consumable by the script.

4 Results

Development was done in a private Ethereum network deployed in two computers. The code has a set of scripts and documentation on how to recreate an Ethereum private network. The final tests were done in Ethereum official test network (Ropsten). There are 3 Ethereum test networks. Two of them use an alternative to Proof-of-Work called Proof-of-Authority where only certain nodes can mine transaction in a semi trusted environment that is not as energy consuming. We use Ropsten which mimics Ethereum current live network.

Once deployed on the test net we recorded the gas costs for each of the operations on our contract execution. We could run all the function inside the contract except for casting a vote which would exceed the amount of gas that the test-net or the main-net allow due to avoid DDoS attacks. Casting a vote requires the contract to verify a ring signature. Verifying a ring signature is linear to the number of public keys in the ring and requires hashing into an elliptic curve. There are algorithms that can hash into an elliptic curve in deterministic polynomial time [13]. Due to the scope of the project we used the 'try-and-increment' method which is a probabilistic algorithm that does a trial and error, each trial requires the computation of a modular square root. All of these computations are very intensive for a computing environment such as Ethereum.

Tallying up the results will also run out of gas, but the tallying does not need to happen on the blockchain, once the secret key has been reconstructed anybody can call the tally function locally and get the result of the election.
We present the costs in gas an in US dollars of running our election contract with different sub-rings sizes. Although we could not cast a vote on the test network because of the computational complexity of verifying a linkable ring signature on the blockchain, we calculated its gas costs separately.

Operation	Gas Cost	USD cost
Setup Contract	1001366	$0.03644
Registration	199760	$0.00728
Cast Vote (SR - 5)	7097929	$0.257
Cast Vote (SR - 15)	24132958	$.879
Cast Vote (SR - 30)	46846331	$1.707

We used the avegae cost of (1 Gwei) per gas to calculate the cost to USD.

5 Conclusions

In this paper, we presented a scheme for running an election on top of a blockchain so that any interested party can have cryptographic assurance of the outcome of the election. We did this by implementing a smart contract on top of Ethereum that enforces the correct execution of our voting scheme. Although our contract can't currently cast votes on the Ropsten test network or the Ethereum test network, we have implemented a blockchain voting scheme that could handle elections. This also shows us some of the current limitation by the existing blockchain infrastructure.

While the idea of blockchain brings us an ideal decentralized trustless environment where we can host important human processes in which we wish to minimize the trust we place on individuals and institutions, it is still far away from being the robust infrastructure needed to solve world problems. Nevertheless, given the spark of its inception hasn't even marked the decade there are probably still a lot of interesting advances to see in the coming years.

In future work, we will investigate into other types of ring signatures such as Borromean ring signatures. And how to use Ethereum as a dedicated computing environment on a separate network just for elections. Given the nature of Ethereum open source community, bugs and security issues are brought to light quickly, this makes it attractive from a security perspective. However, the Ethereum main network can become very confectioned with the daily flow of financial transaction and existing smart contract execution. A future research would look at how to run an election in the cloud, where anybody can deploy a node in a Proof-of-Authority network where we can trust certain institutions to mine votes but anybody can be listening and relaying. The block size could be increased and the gas limits could be raised to make this a feasible system.

References

[1] David L. Chaum. Untraceable electronic mail, return addresses, and digital pseudonyms. *Commun. ACM*, 24(2):84–90, February 1981.

[2] Adi Shamir. How to share a secret. *Commun. ACM*, 22(11):612–613, November 1979.

[3] Josh Benaloh. Simple verifiable elections. In *Proceedings of the USENIX/Accurate Electronic Voting Technology Workshop 2006 on Electronic Voting Technology Workshop*, EVT'06, pages 5–5, Berkeley, CA, USA, 2006. USENIX Association.

[4] F. Hao, P. Y. A. Ryan, and P. Zielinski. Anonymous voting by two-round public discussion. *IET Information Security*, 4(2):62–67, June 2010.

[5] Patrick McCorry, Siamak Shahandashti, and Feng Hao. A smart contract for boardroom voting with maximum voter privacy. 01 2017.

[6] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system http://bitcoin.org/bitcoin.pdf. 11 2008.

[7] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger https://ethereum.github.io/yellowpaper/paper.pdf.

[8] Joseph K. Liu, Victor K. Wei, and Duncan S. Wong. Linkable spontaneous anonymous group signature for ad hoc groups. In *In ACISP'04, volume 3108 of LNCS*, pages 325–335. Springer-Verlag, 2004.

[9] Kyle Croman, Christian Decker, Ittay Eyal, Adem Efe Gencer, Ari Juels, Ahmed Kosba, Andrew Miller, Prateek Saxena, Elaine Shi, Emin Gün Sirer, Dawn Song, and Roger Wattenhofer. *On Scaling Decentralized Blockchains*, pages 106–125. Springer Berlin Heidelberg, Berlin, Heidelberg, 2016.

[10] Loi Luu, Viswesh Narayanan, Chaodong Zheng, Kunal Baweja, Seth Gilbert, and Prateek Saxena. A secure sharding protocol for open blockchains. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 17–30, New York, NY, USA, 2016. ACM.

[11] David Chaum and Eugène Van Heyst. Group signatures. In *Proceedings of the 10th Annual International Conference on Theory and Application of Cryptographic Techniques*, EUROCRYPT'91, pages 257–265, Berlin, Heidelberg, 1991. Springer-Verlag.

[12] Ronald L. Rivest, Adi Shamir, and Yael Tauman. How to leak a secret. In Colin Boyd, editor, *Advances in Cryptology — ASIACRYPT 2001: 7th International Conference on the Theory and Application of Cryptology and Information Security Gold Coast, Australia, December 9–13, 2001 Proceedings*, pages 552–565, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

[13] Thomas Icart. How to hash into elliptic curves. In Shai Halevi, editor, *Advances in Cryptology - CRYPTO 2009: 29th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 16-20, 2009. Proceedings*, pages 303–316, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.