Non-Interactive Bias-free Voting Within ZK Multisig Wallets

Mariia Zhvanko, Artem Chystiakov
Distributed Lab

July 2024

1 Abstract

The paper proposes a practical approach to implement a non-interactive bias-free voting protocol by extending the ZK Multisig Wallet Proposal [KC24]. Without diminishing users' privacy, the extension enables participants' votes to remain private until all the multisig members have voted, leveraging aggregated ECC ElGamal encryption. The Distributed Key Generation (DKG) is used to achieve non-interactiveness in the keys deduction and elimination of centralized, trusted entities. The described approach can be used in multisig wallets with no more than several hundred participants.

2 Motivation

The transparency of public blockchains offers a multitude of advantages, including enhanced traceability of actions, execution verifiability, and openness of data that is available to everyone. However, from a voting perspective, it poses unique challenges in maintaining vote confidentiality and preventing bias influenced by other parties' positions – fundamental aspects of a secure and impartial voting system.

The goal is to create a non-interactive protocol that will provide voters with the assurance that their ballots are secret and their choices are not influenced by how early participants vote.

The solution will allow users to create proposals for a specified number of participants, and vote for or against the proposals without knowing the individual votes or the result until all the users from the membership list have cast their ballots.

3 Specification

3.1 Application flow

The basic application flow heavily relies upon the ZK Multisig Wallet Proposal [KC24] and is described in the context of its entities. The multisig proposal has been extended with ballot encryption and the capability to vote for or against instead of simply approving the proposal.

The flows for proposal creation and casting a vote on the proposal are provided.

3.1.1 Proposal creation

The proposal creation flow is depicted in the following diagram:

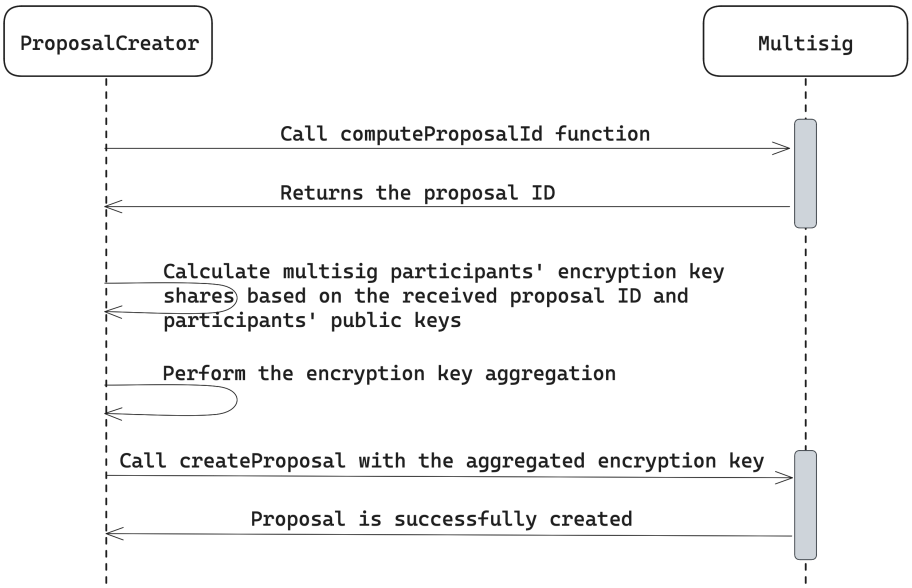

Figure 1: Multisig proposal creation flow

1. The flow starts with the proposal creator computing the ID of the proposal to be used in the generation of a one-time aggregated encryption key.
2. The creator calculates the encryption key share of every multisig participant using KDF based on the proposal ID and the participant's babyJubJub public keys.
3. After receiving all the encryption key shares the creator aggregates them into the final encryption key.

4. The proposal creator invokes the `createProposal` function, providing the aggregated key to be further used for votes encryption.

5. After the proposal is created multisig participants can vote on it.

3.1.2 Voting on the proposal

The voting on the proposal flow is depicted in the following diagram:

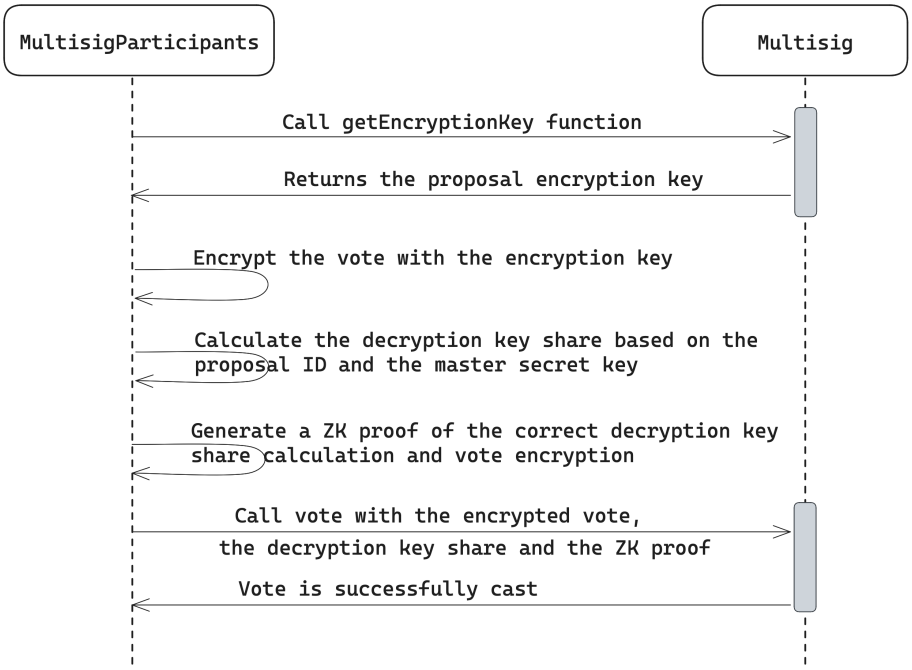

Figure 2: Voting on the proposal flow

1. The user fetches the encryption key of the specific proposal previously created.

2. The received encryption key is used to encrypt their vote.

3. Based on the proposal ID the user calculates the decryption key share using their babyJubJub private key and KDF.

4. After calculating all the needed data, the multisig participant invokes the `vote` function, providing the encrypted vote, decryption key share, and the ZK proof.

5. The smart contract adds the provided decryption key share to the aggregatable final decryption key and sets the vote as successfully cast.

3.1.3 Vote Revelation and Proposal Execution

Only when the last participant has voted the decryption key is complete and can be used to reveal the voting outcome. This is done by calling the `reveal` function. It decrypts the aggregated votes and changes the proposal status according to the voting result. If the majority has voted "for" the proposal status is set to "accepted" and can be executed, "rejected" otherwise.

The diagram below illustrates the votes revelation process encapsulating the decryption and execution logic into the single function `revealAndExecute`.

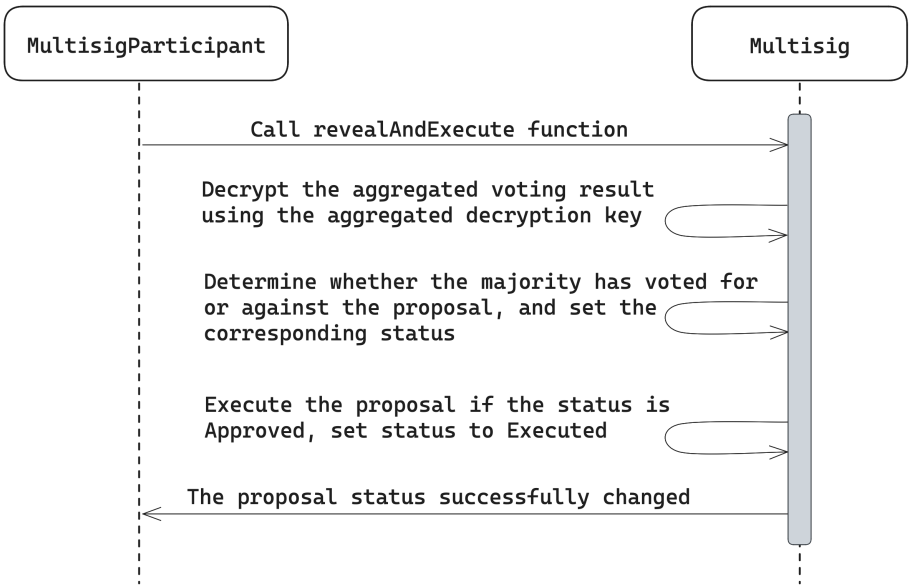

Figure 3: Vote revelation and proposal execution flow

3.2 Functionality

This section outlines the technical features that must be added to the smart contracts, circuits, and SDK to support the protocol described in this paper.

3.2.1 Elliptic Curve Arithmetic Operations

In the context of this document, certain operations are performed on elliptic curves and involve specific mathematical operations distinct from conventional arithmetic:

- Point addition $P + Q$, where P and Q are points on the elliptic curve, is performed according to the defined group operation logic.

- Scalar multiplication $k \times P$, where k is a scalar and P is a point on the elliptic curve, involves adding the point P to itself k times.
- Point subtraction $P - Q$, where P and Q are points on the elliptic curve, is identical to $P + (-Q)$, where $-Q$ is the inverse of point Q.

3.2.2 KDF for Encryption Keys

The protocol utilizes the stealth key schema [CM17] as the deterministic KDF. Such schema enables the generation of unique key shares to encrypt and decrypt votes for each proposal out of the master keys.

The master key pair must satisfy the following:

$$pk = sk \times G,$$

where pk — master public key;
sk — master secret key;
G — base point on the elliptic curve.

The babyJubJub key pair introduced in the ZK Multisig Wallet proposal is used as the master key pair from which the ElGamal encryption and decryption keys are derived. The key derivation procedure is the following:

According to the ZK Multisig Wallet proposal, the `proposalId` and `challenge` are deterministically calculated hash values:

```
proposalId = keccak256(abi.encode(target, value, data, salt))

challenge = poseidon(uint248(keccak256(abi.encode(chainid,
    zkMultisigAddress, proposalId))))
```

Let r be equal to the `challenge` and R be the `challenge` point:

$$R = r \times G$$

The encryption key share is derived as follows:

$$P_i = poseidon(r \times pk) \times G + pk$$

Correspondingly, the decryption key share is derived as follows:

$$x_i = poseidon(sk \times R) + sk \mod n,$$

where n — order of the elliptic curve.

The consistency of the key derivation scheme can be proved by:

$$P_i = x_i \times G = (poseidon(sk \times R) + sk) \times G =$$

$$= poseidon(sk * r \times G) \times G + sk \times G = poseidon(r \times pk) \times G + pk$$

3.2.3 ECC ElGamal Encryption Scheme

The protocol utilizes the Elliptic Curve Cryptography (ECC) modification of the ElGamal encryption scheme [Kob87] to encrypt and hereinafter decrypt the multisig votes.

The aggregated encryption key P is a point on the elliptic curve computed by summarizing all encryption key shares:

$$P = \sum_{i=1}^{N} P_i$$

where N — number of the multisig participants;
P_i — encryption key share (elliptic curve point).

The participant vote is mapped to a point M on the elliptic curve. The generator point G is used as the "for" vote and the point at infinity as the "against". A random value k satisfying $0 < k < n$ is chosen. Afterward, the ciphertext $(C1, C2)$ is computed:

$$C_1 = k \times G$$

$$C_2 = M + k \times P$$

To decrypt the vote, first compute the aggregated decryption key share:

$$x = \sum_{i=1}^{N} x_i \mod n$$

where n — order of the elliptic curve;
x_i — decryption key share (scalar).

Then use the computed aggregated decryption key x to recover the message point M:

$$M = C_2 - x \times C_1$$

The consistency of the key aggregation within the ECC ElGamal scheme can be proved as follows:

$$P = \sum_{i=1}^{N} P_i = \sum_{i=1}^{N} x_i \times G$$

$$D = x \times C_1 = \sum_{i=1}^{N} x_i \times C_1 = \sum_{i=1}^{N} x_i * k \times G = k \times (\sum_{i=1}^{N} x_i \times G) = k \times P$$

$$M = C_2 - D = M + k \times P - k \times P$$

3.2.4 Homomorphic Aggregation of Votes

Using point G and point at infinity as votes makes it possible to form the cumulative voting result homomorphically, summing up the encrypted votes during each vote cast:

$$SC_1 = \sum_{i=1}^{N} C_1 i$$

$$SC_2 = \sum_{i=1}^{N} C_2 i$$

Decrypt the sum to get the aggregated result T:

$$T = \sum_{i=1}^{N} M_i = SC_2 - x \times SC_1$$

The decrypted total T is equal to $v \times G$, where v is the total number of voters who voted "for" the proposal.

To reveal the votes, a multisig participant must loop through the possible scalar values v_i, where $0 \leq v_i \leq N$, off-chain to find the value that satisfies the equation:

$$v_i \times G = T$$

Then they will submit this value to the smart contract where the above equation is checked.

- If $v \leq N/2$, the majority has voted "against" the proposal and its status is set to "rejected";
- If $v > N/2$, the majority has voted "for" the proposal and its status is set to "accepted".

3.2.5 Encryption Key On-Chain Calculation

When the proposal gets created it is essential to check that the aggregated encryption key was computed correctly from the individual public keys and the challenge.

As this value is publicly computable, it can be evaluated on the smart contract by looping through the participants' master public keys and deriving their encryption shares. However, this approach is suboptimal and can be improved by introducing a cumulative public key, which represents the sum of all participants' master public keys. This key is stored on the smart contract and has to be updated whenever the membership list gets updated.

Instead of calculating the encryption key share for each participant individually:

$$P_i = poseidon(r \times pk_i) \times G + pk_i,$$

It is enough to calculate only the hash part and add it to the sum of previous hashes, reducing the number of operations inside the loop:

$$sumHash \mathrel{+}= poseidon(r \times pk_i)$$

Then the aggregated encryption key can be computed as follows:

$$aggKey = sumHash \times G + cumulativePk$$

3.2.6 ZKMultisig

The smart contract interface needs to be extended with the following functions:

```
function getEncryptionKey(uint256 proposalId) external view
    returns (uint256[] memory);

function votesRevealed(uint256 proposalId) external view returns (bool);

function reveal(uint256 proposalId, uint256 approvalVoteCount) external;
```

The `reveal` and `execute` functions may be called together in a single transaction to enhance user experience via `revealAndExecute`. The function should check the success of the votes decryption and continue with proposal execution in that case only.

```
function revealAndExecute(uint256 proposalId, uint256 approvalVoteCount)
    external;
```

The `ProposalStatus` enum also needs to be extended to cover the case when the majority of the participants have voted "against" the proposal:

```
enum ProposalStatus {
        NONE,
        VOTING,
        ACCEPTED,
        REJECTED,
        EXPIRED,
        EXECUTED
}
```

Essential changes regarding the decryption key share need to be applied to the `vote` function:

```
function vote(uint256 proposalId, bytes calldata encryptedVote,
    uint256 decryptionKeyShare, ZKParams calldata proofData);
```

It is also crucial to verify ElGamal encryption key aggregation inside the `createProposal` function for the given proposal. The smart contract has to loop through the active members' master public keys, deriving their encryption key shares, which are subsequently aggregated into the encryption key (see section 3.2.5).

3.2.7 Circom circuits

Every parameter in this paper is provided with the intent to be provable and compatible with zero knowledge circuits. The new list of circuit signals is the next:

Public signals:

- User blinder (output);
- Decryption key share (output);
- Encryption point $C1$ (output);
- Encryption point $C2$ (output);
- Aggregated encryption key (input);
- Proposal challenge (input);
- SMT root (input).

Private signals:

- Master secret key;
- Master public key [optional];
- EdDSA master signature of the challenge;
- The actual vote: point G or inf;
- The random encryption value k;
- SMT inclusion proof.

Utilizing these signals, the circuit must have the following constraints:

1. The provided master secret key is indeed the secret key of the provided master public key.
2. The master public key belongs to the SMT, anchoring to the SMT root.
3. The EdDSA master signature of the challenge verifies to the master public key.

4. The poseidon hash of the signature is equal to the user blinder.
5. The decryption key share is calculated correctly given the master secret key and the proposal challenge (see section 3.2.2).
6. The provided vote is either equal to G or inf.
7. The encryption has been performed correctly given the aggregated encryption key, the random encryption value k, and the vote (see section 3.2.3).

4 Rationale

The ECC modification of the ElGamal encryption scheme [Kob87] was selected for its compatibility with key derivation functions (KDFs) and key aggregation. ElGamal's non-deterministic nature introduced by the random value k guarantees that each encryption operation produces a unique ciphertext, enhancing security.

To mitigate centralization risks associated with traditional decryption key management, like in the Shamir Secret Sharing (SSS) scheme where the existence of a decryption key before secret sharing is required, the Distributed Key Generation (DKG) approach was taken. It provides the mechanism to generate the decryption key in a decentralized way by every participant so that no single party possesses the complete key before revelation.

Most DKG protocols remove the need for a trusted party by employing Verifiable Secret Sharing (VSS), which mandates participant interaction to verify share validity. This process is replaced by verifying ZK proofs of decryption key shares on demand when casting the vote.

Although DKG eliminates the need for key share exchange between parties, it still requires publishing encryption key shares during proposal creation. Using a deterministic KDF, participants can avoid interactive processes, enabling the proposal creator to aggregate the final encryption key asynchronously and independently.

5 Limitations

There are a few limitations inherent to the protocol that need to be considered:

- Participant Removal Deadlock: if a participant to be removed does not vote (which is natural to them), the removal proposal expires and the participant remains a multisig member. A possible solution may be to vote openly on such kind of proposals, e.g. to use the regular ZK Multisig Wallet [KC24] voting procedure.
- Modification of Participant List: it is challenging to add or remove participants from the Multisig Wallet whenever there are active (ongoing)

proposals. Doing this may cause inconsistency in the keys used within the encryption scheme. A possible solution may be to track such proposals and allow their creation only when nothing else is in progress.

- Scalability: votes revelation and proposal results calculation complexity scales linearly with the number of participants. The number of multisig members is recommended to be under 500.

References

[KC24] Oleh Komendant and Artem Chystiakov. *ZK Multisig Wallet Proposal*. 2024. URL: https://distributedlab.com/whitepaper/zk_multisig.pdf.

[CM17] Nicolas T. Courtois and Rebekah Mercer. *Stealth Address and Key Management Techniques in Blockchain Systems*. 2017. URL: https://www.scitepress.org/papers/2017/62700/62700.pdf.

[Kob87] Neal Koblitz. *Elliptic Curve Cryptosystems*. 1987. URL: https://www.ams.org/journals/mcom/1987-48-177/S0025-5718-1987-0866109-5/S0025-5718-1987-0866109-5.pdf.