ZK Multisig Wallet Proposal

Oleh Komendant, Artem Chystiakov
Distributed Lab

May 2024

1 Abstract

The paper proposes an approach to implement a Zero Knowledge (ZK) EVM-based multi-signature wallet to preserve privacy yet not compromise functionality.

The solution consists of 3 components: Solidity smart contracts, Circom circuits and TypeScript SDK.

Smart contracts play the roles of wallets factory and wallets that users interact with to create multisigs and execute collective proposals. Circom circuits, responsible for checking users belonging to the multisig sets and verifying their decisions on whether to execute a proposal or not. TypeScript SDK that couples everything together, allowing seamless integration for any Dapp front end.

2 Motivation

The open nature of the Ethereum blockchain has its pros and cons. On the one hand, Ethereum's publicity enables it to support Turing-complete smart contracts, acting as an ultimate sandbox for Dapps and DEFI. Yet, on the other hand, users transactions, actions, and decisions can be traced and used for abuse.

Frankly speaking, it is possible to take the best from that publicity but preserve privacy via ZK. The goal is to create a simple, permissionless multisig wallet that doesn't disclose anything about its members.

The wallet would allow users to be included/excluded from the members list, the configuration of the "signature threshold", and execution of collectively approved transactions, with zero compromises in privacy.

3 Specification

3.1 Application flow

Before proceeding with the technical deep-dive it is essential to see the high-level picture and understand the basic application flow. The flows for a multisig

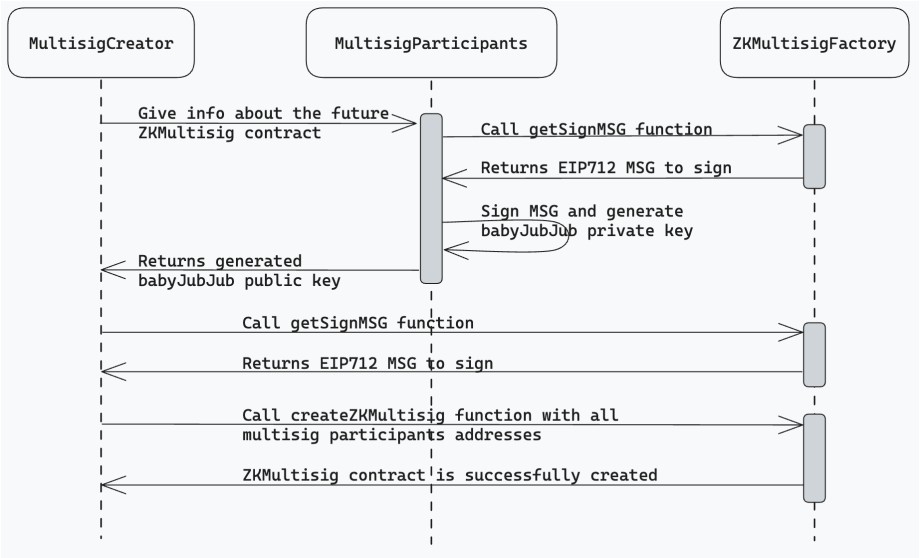

Figure 1: ZKMultisig contract creation flow

wallet creation and generic multisig transaction execution are provided.

3.1.1 Multisig creation

The cornerstone of the application is the multisig wallet. Upon interacting with the application, users create multisig wallets with the list of permitted voters for their business logic.

The multisig creation flow is depicted in the following diagram:

1. The creation flow starts with the user (wallet creator) gathering the public keys of the voters to be added to the permitted list. Since the multisig utilizes ZK to maintain privacy, all the users have to create a special babyJubJub key pair that will be used as their unique identifier before using the application.

   To create these keys, a user signs an EIP-712 structured message with their Ethereum (ECDSA) private key and hashes the obtained signature. The resulting hash is the babyJubJub private key.

   Users may choose between signing the unique messages to get unique public keys for every multisig they are willing to participate in (increases privacy) or using the "default" message to stick to a single public key to be utilized across the platform (possibly better UX).

2. Having acquired all the necessary babyJubJub public keys, the wallet creator invokes the wallet creation function on the `ZKMultisigFactory`

smart contract, providing all the public keys to be added to the permitted list.

Under the hood, the multisig stores the participants in a Sparse Merkle Tree (SMT) data structure, enabling ZK-provable membership proofs and cheap list maintenance. The SMT data structure supports both addition and removal operations, which is a perfect fit for the required logic.

3. After the multisig wallet is deployed, its members can create proposals and vote for them by generating ZK proofs of membership and applying EdDSA blinders for decision non-reusability.

With the described approach we can achieve full privacy for the users by abstracting their real "wallet address" with a babyJubJub one through a deterministic key derivation function (KDF) and decrease the probability of determining the decision-making address from 1 to 1/N, where N is the number of multisig members.

3.1.2 Multisig proposals

The basic interaction flow with the multisig wallet is through the creation of proposals. Having the list of available proposals, wallet users may choose which ones to support via voting. A proposal is set to be accepted if the required quorum of signatures (transactions) is reached.

The multisig proposal creation flow is depicted in the following diagram:

1. The proposal creator (a user from the multisig permitted list) logs in to the application by deterministically recovering the babyJubJub key pair from the "multisig wallet creation" step.

   The KDF algorithm remains the same. The EIP-712 structured message is obtained from the `ZKMultisigFactory`, then signed, and the signature hashed to calculate the babyJubJub private key.

2. The SMT inclusion (Merkle) proof is fetched from the `ZKWallet` to indicate that the user is a member of the multisig.

3. The user generates a ZK proof that permissionlessly verifies that:

   (a) The user belongs to the multisig members list via SMT inclusion proof.

   (b) The user signed (with their babyJubJub private key) the proposal they are supporting.

   (c) The user didn't previously vote for the same proposal through a blinder check of the hash of their EdDSA signature.

4. The number of users "for" that proposal is incremented. If the number exceeds the "signature threshold", the proposal is set to be "accepted" and can be executed.

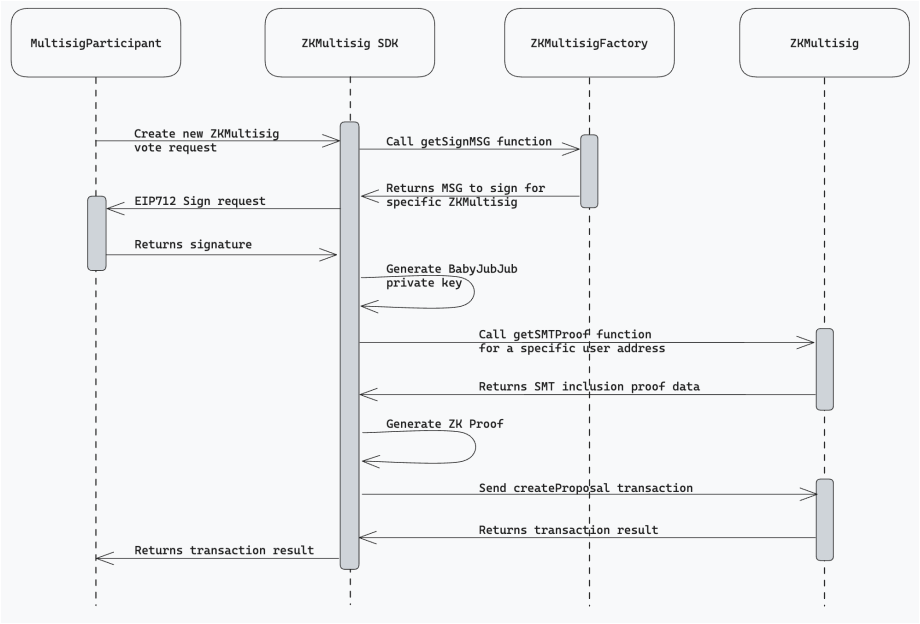

Figure 2: Multisig proposal creation flow

3.2 Functionality

In this section the technical description of smart contracts and circuits is provided. The section outlines the required functionality to be supported by the components to implement the workable prototype further.

3.2.1 ZKMultisigFactory

There are two contracts in the application: `ZKMultisigFactory` and `ZKMultisig`. The factory is used to create multisig wallets and generate EIP-712 messages that are required for the key derivation procedure. `ZKMultisig` is the implementation of the wallet itself.

The `ZKMultisigFactory` interface is defined as follows:

```
interface IZKMultisigFactory {
    event ZKMultisigCreated(
        address indexed zkMultisigAddress,
        uint256[] initialParticipants,
        uint256 initialQuorumPercentage
    );

    function createZKMultisig(
        uint256[] calldata participants,
        uint256 quorumPercentage,
```

```
        uint256 salt
    ) external returns (address);

    function getZKMultisigsCount() external view returns (uint256);

    function getZKMultisigs(uint256 offset, uint256 limit) external view
        returns (address[] memory);

    function computeZKMultisigAddress(
        address deployer,
        uint256 salt
    ) external view returns (address);

    function getKDFMSGToSign(address zkMultisigAddress) external view
        returns (bytes32);

    function getDefaultKDFMSGToSign() external view returns (bytes32);

    function isZKMultisig(address multisigAddress) external view returns
        (bool);
}
```

`ZKMultisigFactory` does not deploy the wallets directly, rather the ERC-1967 proxies are deployed. Moreover, the `create2` approach is taken to provide determinism to the wallets addresses in order to establish the KDF messages upfront.

The salt in the create2 is defined as follows:

```
realSalt = keccak256(abi.encode(msg.sender, salt))
```

The `quorumPercentage` is a decimal value with 10**25 points of precision. 100% is defined as 100_000_000_000_000_000_000_000_000_000. The signatures quorum can be calculated as:

```
max(participants.length() * quorumPercentage / 100%, 1)
```

Users may decide which KDF message to sign:

- The unique message per wallet (increases privacy)
- The default one (possibly better UX)

The structure of the returned messages can be found in section 3.2.5.

The factory also provides a convenient interface for retrieving the list of multisig addresses and checking whether an address is indeed multisig. The design is such to maintain the full on-chain requirement. No back end is needed to operate the application.

3.2.2 ZKMultisig

`ZKMultisig` is a contract that implements the multisig functionality. The implementation includes the management of multisig participants, quorum settings, proposals creation, voting, and their execution. Also, there is a handful of view methods for seamless backendless integration.

The `ZKMultisig` interface is defined as follows:

```
import
    "@solarity/solidity-lib/libs/data-structures/SparseMerkleTree.sol";

interface IZKMultisig {
    enum ProposalStatus {
        NONE,
        VOTING,
        ACCEPTED,
        EXPIRED,
        EXECUTED
    }

    struct ZKParams {
        uint256[2] a;
        uint256[2][2] b;
        uint256[2] c;
        uint256[] inputs; // 0 -> blinder, 1 -> challenge, 2 -> SMT root
    }


    struct ProposalContent {
        address target;
        uint256 value;
        bytes data;
    }

    struct ProposalInfoView {
        ProposalContent content;
        uint256 proposalEndTime;
        ProposalStatus status;
        uint256 votesCount;
        uint256 requiredQuorum;
    }

    event ProposalCreated(uint256 indexed proposalId, ProposalContent
        content);
    event ProposalVoted(uint256 indexed proposalId, uint256
        voterBlinder);
    event ProposalExecuted(uint256 indexed proposalId);

    function addParticipants(uint256[] calldata participantsToAdd)
        external;
```

```
    function removeParticipants(uint256[] calldata participantsToRemove)
        external;

    function updateQuorumPercentage(uint256 newQuorumPercentage)
        external;

    function create(
        ProposalContent calldata content,
        uint256 duration,
        uint256 salt,
        ZKParams calldata proofData
    ) external returns (uint256);

    function vote(uint256 proposalId, ZKParams calldata proofData)
        external;

    function execute(uint256 proposalId) external;

    function getPerticipantsSMTRoot() external view returns (bytes32);

    function getParticipantsSMTProof(bytes32 publicKeyHash) external
        view returns (SparseMerkleTree.Proof memory);

    function getParticipantsCount() external view returns (uint256);

    function getParticipants() external view returns (bytes32[] memory);

    function getProposalsCount() external view returns (uint256);

    function getProposalsIds(uint256 offset, uint256 limit) external
        view returns (uint256[] memory);

    function getQuorumPercentage() external view returns (uint256);

    function getProposalInfo(uint256 proposalId) external view returns
        (ProposalInfoView memory);

    function getProposalStatus(uint256 proposalId) external view returns
        (ProposalStatus);

    function getProposalChallenge(uint256 proposalId) external view
        returns (uint256);

    function computeProposalId(ProposalContent calldata content, uint256
        salt) external view returns (uint256);

    function isBlinderVoted(uint256 proposalId, uint256 blinderToCheck)
        external view returns (bool);
}
```

The multisig wallet intentionally omits the incremental enumeration of proposals to prevent ZKPs replay and frontrunning attacks. This is achieved by asking the proposal creator to sign the challenge of the proposal they are creating.

The proposal id and the challenge is deterministically calculated as follows:

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

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

Critically, the proposal creator will automatically vote for the proposal upon creation.

The wallet possesses several "wallet management" functions that require `onlyThis` modifier and should only be called through proposals. These are `addParticipants()`, `removeParticipants()`, and `updateQuorumPercentage()` functions.

There is no "voting against" functionality, all the voters intrinsically vote "in favor". Once the quorum of signatures is reached, the proposal can be further executed (by anyone). Also, the proposal expires if the quorum is not reached in the allocated time.

3.2.3 Circom circuits

The Circom circuits are the beating heart of the application. They drive privacy and enable vital checks to be permissionlessly performed. The circuits are tightly integrated with the smart contracts and come as a complete module.

Circuits check if the user's account belongs to the SMT of all users of the multisig, verify the EdDSA signature of a proposal challenge to prove the ownership of the particular account, and generate a deterministic blinder based on the passed signature.

The blinder is a poseidon hash of the passed signature and it is used on the contracts to check that the user is not "double voting". Hashing the signature is safe due to the deterministic nature of EdDSA. There are no known EdDSA "signature flipping" attacks, so as long as the message being signed is unique, using signature as nonce does not impact security.

Circuit private signals are:

- EdDSA signature of the challenge.
- SMT Inclusion proof.

Circuit public signals are:

- User blinder (output).
- Proposal challenge (input).

- SMT root (input).

3.2.4 KDF

Key Derivation Function (KDF) is such a function that deterministically creates a babyJubJub private key from some input. In our case, the input is the Ethereum ECDSA signature of the EIP-712 formatted message.

The KDF is defined as follows:

```
message = getKDFMSGToSign() or getDefaultKDFMSGToSign()
signature = eth_signTypedData_v4(message)
privateKey = keccak256(keccak256(signature))
```

Note that It is crucial to never reveal the signature as the private key derives from it directly.

3.2.5 KD EIP-712 message

To create a key derivation EIP-712 message, a KDF message typehash is used that includes the `ZKMultisig` address of the contract. This is sufficient as the network and the contract information is included in the standard EIP-712 domain structure.

```
bytes32 KDF_MSG_TYPEHASH = keccak256("KDF(address zkMultisigAddr)");

bytes32 kdfStructHash = keccak256(abi.encode(KDF_MSG_TYPEHASH,
    zkMultisigAddress));
```

The `getKDFMSGToSign()` and `getDefaultKDFMSGToSign()` functions create an EIP-712 message as described above. The `getDefaultKDFMSGToSign()` function uses a zero address (0x0000000000000000000000000000000000000000) for the construction of the default message.

3.2.6 Relayers

Since the above approach is completely independent of the EVM addresses from which transactions are sent, users can utilize different relayers to preserve anonymity. Protocol-friendly relayers such as GSN can be used, however, that requires additional integration logic in the SDK or the front end.

3.3 Security

There are several security aspects that have to be considered before proceeding with the multisig implementation.

1. Trusted setup. Building on-chain verifiable circuits with Circom would probably demand using Groth16 as a zk-SNARK proving system. Groth16 requires a per-circuit trusted setup that has to be properly carried out.

2. Private key / signature leaks. It is essential to keep the key derivation ECDSA signature private. The babyJubJub key pair is derived directly from it, so leaking the signature would render the multisigs (in which the user is in) vulnerable to phishing / spamming attacks.

3. Proposal frontrunning. Even though the proposal challenge is derived deterministically from the proposal contents it is still possible to frontrun the proposal creation with an identical proposal. This does not directly impact security, yet should be noted.