zkMultisig (v0): scalable offchain voting with onchain trustless binding results

arnaucube[1,2], Pau Escrich[1],
Roger Baig[1,2], Alex Kampa[2]

[1]Vocdoni
[2]Aragon Research

May 2022

WIP: This document is in early stages. Current version: 2022-5-27.

Abstract

Traditional multisig contracts allow the execution of ethereum transactions by a threshold of support of a predefined set of signers. Due block space limitations and increasing gas costs, in practice this is translated to multisigs controled by < 10 users.

The design proposed in this document, aims to allow the execution of ethereum transactions by a threshold of a set of thousands of users, keeping constant gas costs. Additionally we introduce mechanisms such as *weighted voting* into the design.

The main idea, is that the computation and verification of the votes, is computed off-chain, and through a validity proof is proved to a Smart-Contract that everything is correct, similar to a zkRollup (Validity proof). This makes zkMultisig a universal verifiable voting system, which can not only be verified inside any EVM-chain, but by any actor with capacity to compute Pairings.

The contribution of this document is not an ideal design, but a specific design that we implemented using existing zk tools intended to trigger onchain ethereum funds movement. This document is a first step in an exploratory path of using zk schemas for voting.

Contents

1 Introduction

TODO. this section is not yet developed, currently there is a list of content that will be written at some point.

Concepts to be written:

- This document describes a concrete design specifically designed to be implemented with existing tools.
- This is not a theoretical design, but a concrete short-term design with an usable implementation.
- It's far from ideal, it's just a first step towards the direction of using zk for voting
- We're already working in further designs
- Explain the main idea of do the votes verification and results computation offchain, and then verify it onchain through a single zk proof

2 Preliminaries

WARNING: All this section is a big WIP.

This section describes the different cryptographic primitives used in the zkMultisig. The main reason for using these concrete primitives, is because we are constrained into EVM (Ethereum Virtual Machine), which only supports bn254 pairing, and also that we already have them implemented in Circom (used for the R1CS constraint definition as a form of circuits) and in Go (used for the Node).

2.1 zkSNARK

We use Groth16, through Circom, over the bn254 pairing. This constraints the rest of cryptographic primitives that we use (hash functions, merkletree, elliptic curve).

2.2 Hash function

As we're working in the field of the bn254 pairing, its expensive in terms of number of constrains to use traditional hash functions. For this reason, we use a *snark friendly* hash function, more concretely the Poseidon hash function [1].

The benefits are in terms of numbers of constraints, as for example, while the circuit of the Keccak256 implementation takes around $150k$ constraints, the Poseidon implementation for 2 inputs takes around 250 constraints.

More details on Poseidon can be found at poseidon-hash.info.

2.3 Merkle Tree

We use a binary sparse Merkle Tree, where the leaves l are formed by a *key* k and a *value* v. A leaf position in the tree is uniquely determined by the binary representation of its k.

More details on the concrete design of the Merkle Tree used can be found at: iden3/Merkle-Tree.pdf

2.4 Elliptic curve

We need an elliptic curve that voters will use to sign their votes, and in the same way that happens for the hash function, using a traditional elliptic curve non snark-friendly would lead to a large amout of constraints in our circuit. For this reason, we use an elliptic curve which is embedded in the bn254 pairing field. In this way, we can compute the embedded elliptic curve operations in the native field of our constraints system over the pairing field.

As we're using the pairing bn254, we use the embedded curve *BabyJubJub*, and we use the EdDSA algorithm for the signature scheme. A more detailed description of the BabyJubJub can be found at EIP-2494 [2] and in iden3/BabyJubJub.pdf.

Since Ethereum users already have a Metamask account (which works with *secp256k1* elliptic curve), what current zkRollups do to generate the BabyJubJub keys (without needing to store the private key in the browser) is to derive the babyjubjub private key from a secp256k1 signature done with Metamask. The signature specification follows EIP-712 [3].

3 Protocol

3.1 Actors

TODO: a bit of intro/context.

i. *Voters*: users that belong to the *Census* and are elegible to vote.

ii. *Census creator*: builds the census from a set of *PubKs*.

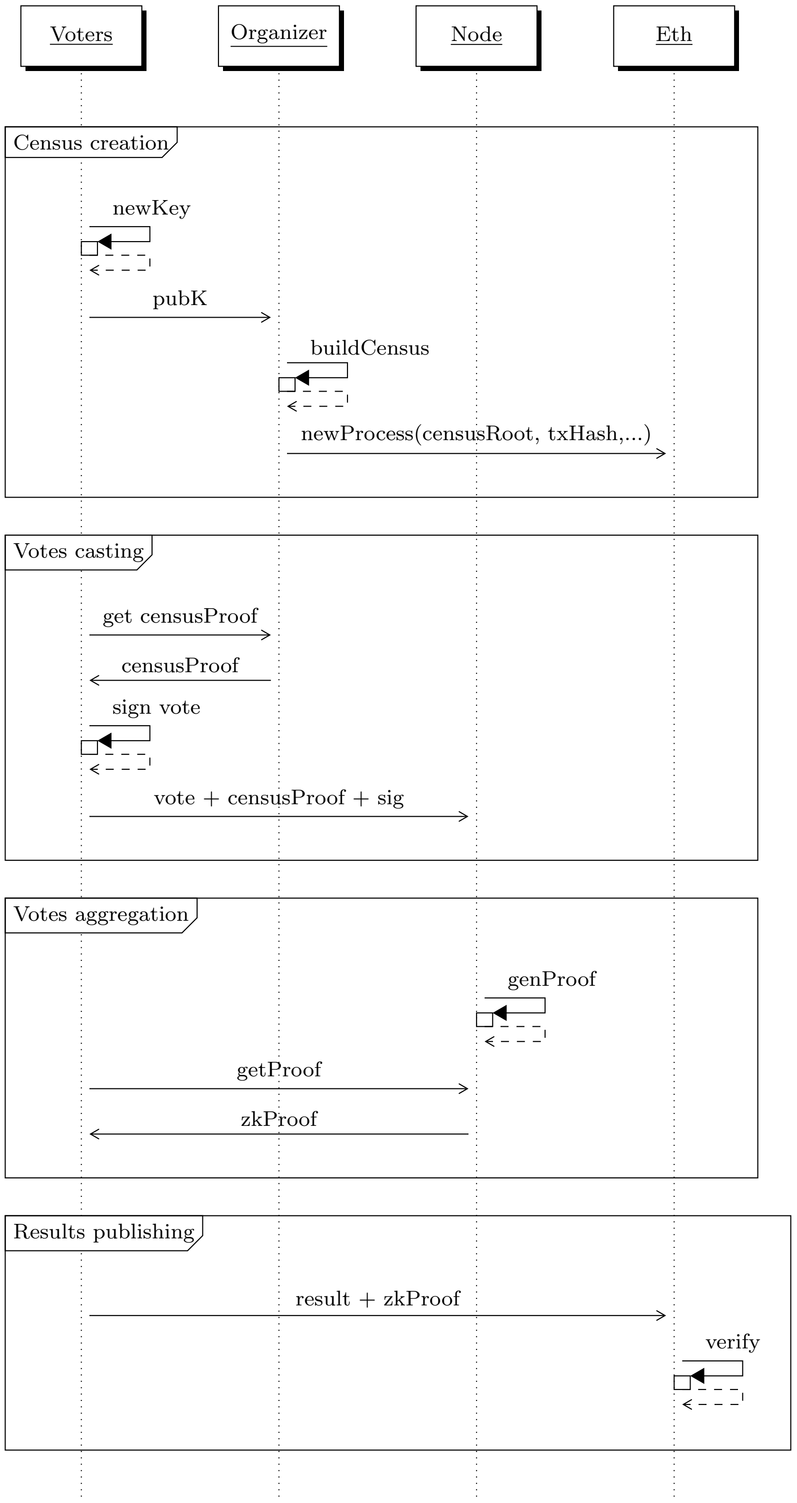
iii. *Process creator*: creates the process with a specific configuration, in most cases will be the same actor as the *Census creator*.

iv. *Node*: receives the votes, and generates the zk proof for the result.

In most of the next sections we will refer to the *Census creator* and *Process creator* as the *Organizer*.

3.2 Overview

There are four main phases: census creation, vote casting, votes aggregation, results publishing.


Voters
Organizer
Node
Eth
Census creation
newKey
pubK
buildCensus
newProcess(censusRoot, txHash,...)
Votes casting
get censusProof
censusProof
sign vote
vote + censusProof + sig
Votes aggregation
genProof
getProof
zkProof
Results publishing
result + zkProof
verify

1. Organizer builds the census
   - Placing the user's *PubKs* and *weights* in the MerkleTree leaves
2. Organizer publishes *census root* (MerkleRoot) in the Ethereum SmartContract
   - And makes the MerkleTree available
3. User retreives their *census proof*
4. User performs the *signature* over their *vote*
5. User sends to the Node their *signature, vote, census-proof*
6. Once the defined *ResultsPublishingStartBlock* is reached, the Node generates the *zk witness* from all the users data
7. Node generates the zkProof
8. Organizer retrieves the zkProof from the Node
   - This can be done by any user
9. Organizer sends the zkProof to the Ethereum SmartContract
   - This can be done by any user
10. SmartContract verifies the zkProof, and triggers the tx

3.3 Contract timeline overview

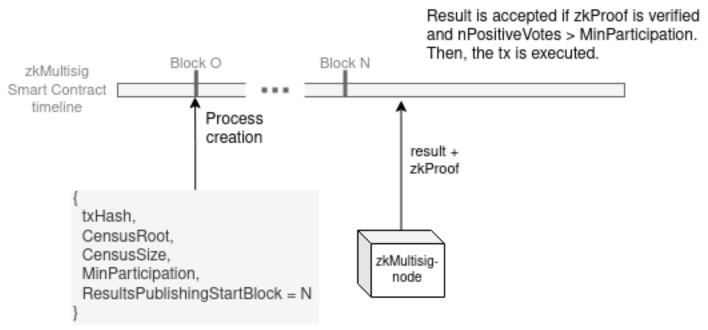
With the current zkMultisig circuit design, there are two possible modalities of usage available. The main difference is at the results publishing phase in how the results are handled by the *Smart Contract*.

This section describes two modalities, the *multisig* and the *referendum*.

3.3.1 Multisig modality

The most straightforward usage of the zkMultisig is by a similar timeline than the traditional multisig mechanism, which consists on gathering the users votes, and as soon as enough support is received, the *result + zkProof* can be sent to the *Smart contract*, which verifies it.

In terms of the zkMultisig, this means that as soon as a *result + zkProof* is published in the *Smart Contract* containing enough support (enough positive votes), and passing the verification, it triggers the funds movement.


Result is accepted if zkProof is verified and nPositiveVotes > MinParticipation. Then, the tx is executed.
zkMultisig Smart Contract timeline
Block O
Block N
Process creation
result + zkProof
{ txHash, CensusRoot, CensusSize, MinParticipation, ResultsPublishingStartBlock = N }
zkMultisig-node

3.3.2 Referendum modality

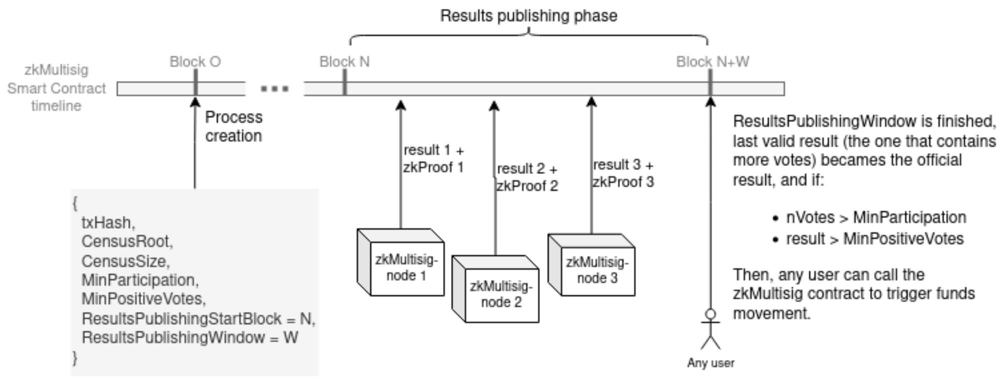
> WIP: maybe for sake of not overcomplicating things, is better to remove this section and all 'referendum' approach from the document, and leave only the 'multisig' approach.

While in the *multisig modality* (3.3.1) the quorum is based on having enough support over a simple threshold, the *referendum* approach allows to take into account the users that want to vote against a proposal (appart from the users who are supporting it).

This requires a bit of a more complex timeline described as follows.

When the *Organizer* creates the new process, specifies not only the *ResultsPublishingStartBlock*, but also the *ResultsPublishingWindow*, which indicates the number of blocks right after the *ResultsPublishingStarBlock* in which the *Smart Contract* will be accepting new *results*.

The rules in which the *Smart Contract* accepts new *results* are that the *zkProof* verifies correctly, and that the *nVotes* (a circuit public input) used in that result is bigger than the previous result published.


Results publishing phase
zkMultisig Smart Contract timeline
Block O
Block N
Block N+W
Process creation
result 1 + zkProof 1
result 2 + zkProof 2
result 3 + zkProof 3
{ txHash, CensusRoot, CensusSize, MinParticipation, MinPositiveVotes, ResultsPublishingStartBlock = N, ResultsPublishingWindow = W }
zkMultisig-node 1
zkMultisig-node 2
zkMultisig-node 3
ResultsPublishingWindow is finished, last valid result (the one that contains more votes) becames the official result, and if:
• nVotes > MinParticipation
• result > MinPositiveVotes
Then, any user can call the zkMultisig contract to trigger funds movement.
Any user

3.4 Circuit

The circuit defines a set of constraints that must be satisfied in order to accept a given result in the *Smart Contract*.

The checks defined by the circuit constrains are:

- Each used *PubK* exists in the *Census Tree* under the public *Census Root*
- Each vote is signed by one of the valid *PubK*
- Each signature is valid
- A *PubK* is not used more than one time in a result
- The result is equal to the addition of all the valid votes, compounded by the *PubK* weight
- All the votes are casted for a specific *ChainID* and *ProcessID* (to prevent proof reusability attacks)
- If enabled, the *receipts* are well computed (more details at section 3.8.2)

TODO

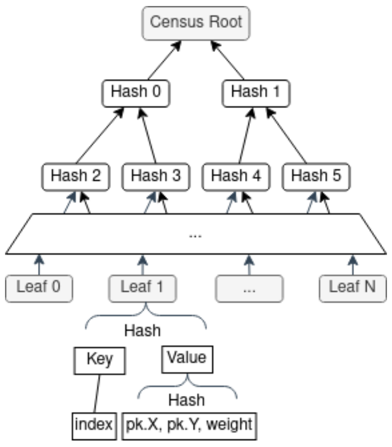
- vote value (0 or 1)
- how the indexes are checked
- maybe a diagram of the circuit
- add the number of constraints of the circuit for 1k, 10k, 100k voters circuits

3.5 Census creation

Each user generates a key pair and sends their *PubK* to the Census creator. Census creator can be the voting process organizer.

The Census creator will add all the user's *PubKs* to a Merkle Tree (from now on, the *Census Tree*), together with its assigned *weight* and *index*. The *weight* determines the voting power of each *PubK*, and the *index* is an self-incremental integer used to determine the position of each leaf in the *Census Tree*.

In the *Census Tree* leaves, the *index* is placed at the *key* of the leaf, and the *PubK* and *weight* are hashed ($H(pk.X||pk.Y||weight)$) and placed as the *value* of the leaf.



As the *Census Tree* is generated from a list of *PubKs* and *weights*, it can be generated by any user in their browser or in a server. Moreover, it could be generated by a *Smart Contract*, this would lead to some further designs where users can deposit tokens into the contract that builds the *Census Tree* out of all the *PubKs* of token depositers, giving to each *PubK* a *weight* based on the amout deposited. We're leaving this out of this initial iteration of the zkMultisig, but a bit of more detail on this idea can be found at 6.

3.5.1 Census proof

The *Census Proof* is a Merkle Proof, which is a *proof of membership* of a leaf in the tree under a given *root*. This proof is formed by the *siblings* in the tree across the path from the leaf to the root. $\boxed{\text{TODO explain more. Comment on } O(n)}$

The same kind of proof is used for the *Vote Receipts*.

3.6 Process creation

Once the census is closed, the Root of the CensusTree becomes the *Census Root*, which is sent by the Organizer to the SmartContract, to indicate the creation of the voting process. Together with the *CensusRoot*, the organizer also sends the following data to the Smart Contract:

- *Census size*: the number of *PubKs* contained in the *CensusTree*.
- *Results publishing start block*: the block number where the SmartContract will start accepting results.
- *Minimum of participation*: the threshold of minimum votes aggregated in a valid result.
- *Tx Hash*: the hash of the tx that is intended to be executed if the voting process succeeds.

3.7 Votes aggregation and results computation

The users send their *votes + censusProof + signatures* to the *node*, which once the process threshold is reached (defined in the *Smart Contract* on process creation), will aggregate all the received votes through the zk proof.

The *node* prepares the *witness* from the votes data and generates the proof, fullfilling the *zkMultisig circuit*.

3.8 Results publication

Once the proof is computed, the node stores it assigned to the process, for any user being able to retreive it. This design is to prevent the node from sending direct transactions to ethereum, needing a wallet with all the security consequences derived from it.

Any user can retreive the proof from the node, and then send it to the *Smart Contract* by paying the gas. The DAOs using the zkMultisig can have a mechanism where the DAO pays back the costs of such transactions to the user.

3.8.1 Multiple nodes

Once the *ResultsPublishingStartBlock* is reached, any user can send to the *Smart contract* the results + zkProof that the *Node* has computed (proving that the result is correctly computated for valid voters). One problem that may happen, is that results *Node* computes a result including only a subset of all the votes, including only the votes that give support to their preference, and not including many votes that would conclude with a different result.

In the *referendum modality* (3.3.2), to prevent this, the zkMultisig *Smart Contract*, accepts not only one result + zkProof publication, but many of them during a period of time (ResultsPublishingWindow), as long as the nVotes of the zkProof is larger than the previous published result + zkProof. This would allow the users to send their votes to multiple zkMultisig-nodes, and then if some of the nodes discards their votes, another of the nodes would have larger number of votes included, and their result will prevail.

In the *multisig modality* (3.3.1) the solution is simplier. The users can send their votes to different available *Nodes*, and the first *Node* to reach the minimum amount of votes can generate a valid proof that will be accepted by the *Smart Contract*. This case is much simplier, because even if the existing *Nodes* are all against a specific voting process and want to censor all the votes supporting it despite there is a majority of voters supporting the proposal, in that case, the people in favor of the proposal succeeding can deploy their own *Node* and direct all their votes there, and as soon as they reach enough votes, can generate a proof and send it to the *Smart Contract*.

3.8.2 Vote receipt

Vote receipts are an optional feature that is set on process creation. With receipts the users can check that their vote has been included in the final result,

consequently, they can prove it to a 3rd party that their vote has been included (without possibility to prove the content of their vote). That's why this feature is set optionally, so if a process does not want to allow reciptness, can be disabled.
.
The way how vote receipts work is the following. When the *Node* aggregates the votes, also it builds a Merkle Tree (from now on *Receipts Tree*), which contains all the *PubKs* and *indexes* of the *votes* that are being aggregated in the proof. The *merkle proof* of inclusion of the set of *indexes* and *PubKs* is used as a private input in the circuit, and the *Receipts Root* (the Merkle Root of the *Receipts Tree*) is used as a public input. In this way, the results publication in the Smart Contract, includes the *Receipts Root*, and the users can check that their vote has been included as their *PubK* is in the *Receipts Tree* under the published *Receips Root*.

The circuit ensures that all the *PubKs* used for the receipts are the *PubKs* that are being used also for the votes computation.

One thing to note, is that the receipts design of this iteration does not allow vote-buying, as the users can not prove the content of their votes, but does allow 'abstention buying', as the voter that didn't vote can prove that their vote has not been included in the final results (*non-membership proof*).

4 Properties

Properties covered for this initial version:

- *Universal verifiability*: because of usage of validity proofs, the results are verfiable by any actor.
- *Unforgeability*: nobody (neither users nor the *Node*) is able to modify the user's votes, neither add fake votes (eg. votes which user is not in the census).
- *Trustless*: because of the previous properties, the system can run in a trustless way, where no human check is needed to ensure the validity of data constructions.
- *Binding execution*: because of its universal verifiability characteristic, the proof verification can trigger onchain actions in a trustless way (eg. moving funds of a DAO), directly from the voting process result (without human intervention).
- *Offchain/gasless voting*: users vote offchain, and the zkMultisig-node aggregates the computation and verification of all the votes, signatures and census-proofs, in a succint zkProof, which is sent to the SmartContract. The only transactions sent onchain are the *process creation* and the *results publishing*.

- *Scalability*: compared to *traditional multisig*, the zkMultisig would scale up to thousands of voters, all aggregated in a single ethereum tx. [TODO: still pending on the final numbers with the real-world circuits]
- Census is *chain agnostic* and *token agnostic*:
  - *Chain agnostic*: the census is build off-chain, and the proof of correct results computation can be published into any EVM chain (furthermore, into any chain that supports Pairing computation). So a zkMultisig census could be used in Ethereum mainnet, but also in other chains.
  - *Token agnostic*: because of the nature of census, it does not depend on tokens holdings, but on a set of *PubKs*.

Properties *not covered* (that may be covered in future designs):

- *No voter privacy*
  - While the relation between votes and publicKeys is not published in the blockchain, the zkMultisig-node will know these relations and could make them public.
  - Future designs could focus on privacy preserving voting solutions.
- *No token-based voting*: as previously mentioned, the scope of zkMultisig is not about token-based voting.

4.1 Frictions of the v0 design

- User friction on census creation
  - Users need to generate their public keys and send them to the voting-process organizer, so the organizer can add them into the census that will be used for the voting process.
  - Once a census is created, it can be reused for as many voting processes as needed.
- Trusted setup ceremony
  - here will go a short explaination of TS concept, for the moment, here are some posts explaining it: [1], [2], [3].
- Not fully decentralized (check section 3.8.1)

5 Implementation

As mentioned earlier, we implemented the proposed design, which consists of:

- Circuits: zkSNARK (Groth16) Circom circuit, which proves the correct offchain computation of the aggregation of valid user votes and result computation.
- Smart contracts: onchain registry of processes, also verifies the zkProofs and results and triggers onchain tx execution.
- Node: zkMultisig node, similar to a zkRollup node. Aggregates the votes and generates the zkProof.
- Client lib: typescript library used in the user's browser, to create keys, signatures and cast the votes.

6 Future work

As we've seen, this document's proposal is a first limited step in the direction of using *validity proofs* for offchain votes computation with onchain verification. That's why we have few features that we're starting to research in order to add in further zkMultisig iterations, and even for completly new unrelated designs.

One of the probably more straightforwards features to be added would be adding homomorphic encryption to the votes, and doing the necessary checks over the encrypted votes inside the circuit, so the content of the votes is not known until the decryption keys are published at the end of the voting process. This would prevent leaking information on the voting trends during the voting process, which could influence the final result.

Some other interesting lines of research are being able to verify *EthStorage-Proofs* (ethereum state patricia merkle tree proofs), to verify inside the circuit that the voter is an Ethereum address that holds a certain amout of tokens. This would be used together with the verification of the *secp256k1* ECDSA (Ethereum signatures). Both verifications in the current constraint system that we use (*Groth16*'s *R1CS*) require a big amount of constraints, and would not be feasible for thousands of users, but it's an interesting line of research for the future.

Another very useful capacity is to verify a *zk proof* inside another *zk proof* (recursion). This can be achieved by different techniques, such as computing the Pairing inside the circuit, or by using order cyclic curves. This capacity is very interesting, because it would allow to let users generate a *zk proof* that they belong to the *Census Tree* without revealing who they are, and then the *Node* would aggregate all those proofs in another *zk proofs*, which is the one that would be verified in the *Smart Contract*. This would enable privacy voting.

An interesting idea to build on top of the zkMultisig, is to set up a Smart Contract that builds a *Census Tree*, where users can deposit ETH or any ERC20 token, and when the deposit is done, the contract adds a new leaf in the *Census Tree* with a *weight* value corresponding to the amount of tokens deposited by the user. In this way, the contract maintains the *Census Tree*, based on the deposits of tokens of the users, and the users can vote on proposals to spend the deposited tokens by using the same approach than the zkMultisig. This could be managed by a contract on top of the zkMultisig contract. In short words, this would be a kind of 'offchain DAO with onchain validity-proofs'.

7 Acknowledgements

TODO. Jordi Baylina, Barry Whitehat

References

[1] Lorenzo Grassi, Dmitry Khovratovich, Christian Rechberger, Arnab Roy, and Markus Schofnegger. Poseidon: A new hash function for zero-knowledge proof systems. Cryptology ePrint Archive, Paper 2019/458, 2019. https://eprint.iacr.org/2019/458.

[2] Eip-2494. https://eips.ethereum.org/EIPS/eip-2494.

[3] Eip-712. https://eips.ethereum.org/EIPS/eip-712.