Aragon Labs is doing some experiments to allow secure off-chain voting that can be verified on-chain without subjective Oracles or any other trusted component.

The mandatory requirements

for this proposal are:

1. Permissionless
2. Anti-censorship
3. Binding results on Ethereum for executing options
4. Gassless for voters
5. No token bridging required
6. As simple as possible; no sidechain involved
7. Usable with ERC20/ERC777 and NFTs for voting

The constraints

of this proposal are:

1. There is no user anonymity, votes can be attached to an Ethereum address
2. The system is not receipt-free, thus vote buying might be possible
3. The implementation of this proposal is not designed to handle national-scale elections, but to be used by DAOs or ERC20/NFT holders. So a maximum number of allowed votes should be set (depending on performance and costs)
4. No incentivization model for relayers is defined

Initial proposal

Under the design of this proposal, when creating a new voting process, organizers submit a transaction to Ethereum specifying the contract address of an ERC20 token to use as a voter census. The Storage Root Hash of this address, at a specified block height, becomes the census root for this process. Anyone who holds the given token can prove their eligibility by providing a storage Merkle proof (via EIP1186) of their balance for the token. They can then cast a vote by sending the proof (siblings) and a signature to a zk-SNARK rollup relayer, which will compute a proof of the final results.

The actor computing the zk-SNARK proof (coordinator) for the results could potentially decide to exclude some votes, thus censorship is possible. We solve this problem by enabling anyone, not just the coordinator, to send new votes. A user can generate and submit a rollup containing just their vote if they detect that a coordinator has not included this vote.

Our proposal uses snarks mainly for the following purposes:

- To verify that an address has not previously voted via Merkle tree accumulator
- To verify that the user has tokens via storage proofs
- To compute partial results on a batch of votes
- To verify the signature of the vote

On this schema, we can identify 2 main problems to solve:

Problem 1: ERC20 storage proof verification is not snark friendly

ERC20 storage proofs are very complex to verify within a snark. This is partly due to storage proofs' use of RLP parsing and multiple keccak256 hash verifications, both of which are inefficient to compute in state-of-the-art snark rollup technology. This problem is difficult to hack around; for the moment we solve this dilemma with optimistic validation.

Problem 2: ecdsa/secp256k1 signature verification is not snark friendly

We need to define a mechanism to verify user signatures. One of the current standards is EdDSA using a BabyJubJub key, which can be derived from an Ethereum signature by using the signature as a raw private key, allowing the user to recover the address. As this method relies on a user signature, it is unfortunately vulnerable to the potential threat of a malicious agent gaming the user- those expecting to sign with a web wallet could be tricked into signing fraudulent transactions. This

vulnerability exists wherever a web wallet is used to sign, and we think a potential solution could be a web3 method to derive a private key using the web address as a derivation path.

There is an additional challenge here of proving that each token owner's Ethereum address approves this BabyJubJub key for voting at the block height of a voting process' creation. We achieve this with a singleton smart contract that maps Ethereum addresses to BabyJubJub public keys, where a user must add their key to the smart contract via a standard transaction. The mapping of an address to a key can be challenged via an optimistic storage fraud proof (since we have already opened the door to the optimistic validation of storage proofs). This solution also solves the data availability problem with a reusable design, as it is expected that these authorized keys will be used multiple times in different voting processes.

In summary, we can handle most verifications, but not all, inside a SNARK.

- To verify that an address has not previously voted via Merkle tree accumulator → SNARK
- To verify that the user has tokens via storage proofs → Optimistic
- To compute partial results of the voting → SNARK
- To verify the signature of the vote → SNARK

A proposal for a PoC

User

- Creates a BabyJubJub key, derived from an Ethereum signature, and registers it to the Voter Registry smart contract
- Retrieves the voting information and the storage proof for their account, generates a signature on the vote package, and forwards it to a relayer or a set of relayers.

Voting smartcontract

- Registers the voting process, including the ERC20 smart contract address, the slot index of the ERC20 address⇒balance mapping, the state root hash for the voting process start block, and the process parameters for computing the vote tally (see the Ballot Protocol Introducing the Vocdoni Ballot Protocol)
- Listens for the registration of new votes via zk-rollup, a snark that proves
- the result calculation
- the vote signature is made by a BabyJubJub key
- the result calculation
- the vote signature is made by a BabyJubJub key
- Keeps the voting accumulators updated
- Keeps the list of voters updated
- Allows anyone to challenge the last vote registration fraud proofs. A challenge must provide one of the following:
- A storage proof that proves that a voter's Ethereum address does not have tokens
- A storage proof that proves that a voter's Ethereum address is not linked to a BabyJubJub key
- A proof that the BabyJubJub key voted (the key is in the "already voted" tree)
- A storage proof that proves that a voter's Ethereum address does not have tokens
- A storage proof that proves that a voter's Ethereum address is not linked to a BabyJubJub key
- A proof that the BabyJubJub key voted (the key is in the "already voted" tree)

Relayer

- phase 0:

the election process is created on Ethereum and a Relayer is selected from a list of available relayers. The election organizer needs to pay for the costs of the election (rewarded to the coordinator). The organizer provides the EVM bytecode that needs to be executed after the election on the DAO smart contract(s) depending on the results.

- phase 1:

voting starts, anyone can send vote packages to the selected coordinator (HTTPs or libp2p transports might be used). The coordinator rolls up the selected results in batches, builds a zk-SNARK proof, and uploads this proof and results to Ethereum. Collects the votes cast by users, verifies them, and broadcasts them to the other relayers

- phase 2:

other coordinators that detect any vote that has not been added can roll up their own votes and send a zk-SNARK validity proof to the voting smart contract. Additionally, if they detect that some vote has been added incorrectly, they can send a fraud proof to prove that the previously added result is invalid, and the coordinator that produced this result will be slashed.

- phase 3:

when the voting date limit is reached

- the sum of the uploaded results (by the coordinator and by any third party) are considered valid, the reward is distributed among the coordinator and the actors that included more votes (if any).
- anybody (usually the coordinator) invokes the EVM bytecode to be executed, using the final results as an input.
- the sum of the uploaded results (by the coordinator and by any third party) are considered valid, the reward is distributed among the coordinator and the actors that included more votes (if any).
- anybody (usually the coordinator) invokes the EVM bytecode to be executed, using the final results as an input.

The circuit and the contract

A zk-SNARK will aggregate a list of cast votes. The zk-SNARK proof is either valid or not based on a given list of votes, a census root, an election identifier (electionId) and an aggregated result.

[

image

726×591 51.2 KB

](https://ethresear.ch/uploads/default/original/2X/d/dc142761b5506766bbd42ebe8bf5ca89bbc003fd.png)

zk-SNARK INPUTS

- hash of inputs (PUB) (this is done to reduce the gas cost of the snark verification)
- electionId (PRIV)
- computation of the voting results of this batch (PRIV)
- current nullifiers root (PRIV)
- updated nullifiers root (PRIV)
- number of votes in the batch (PRIV)
- vote values and corresponding BabyJubJub signatures [1…BATCHSIZE] (PRIV)

The inputs of the smart contract function call for uploading the coordinator results are:

- electionId
- list of voter public keys in this batch
- updated nullifiers root
- computation of the voting results of this batch
- snark proof

Proof of concept implementation

We implemented in https://github.com/vocdoni/voterollup the minimum viable smart contracts and circuits to check the costs and viability of the solution. This PoC only includes the user registry, vote aggregation, and fraud proofs verification.

Our testing incurred the following gas costs:

user_key_registry deployment 258,536 registration 68,956

voting deployment 6,673,159 new_voting 25,989 aggregate_rollup 291,801 fraud_proof_1 574,574 (babyjubjub key not registered) fraud_proof_2 908,822 (account ERC20 balance is zero)

We did some measurements to estimate a feasible number of votes to aggregate, using a standard server with 32GB RAM / 8 CPUS. We found that is possible to aggregate up to 300 votes (with a 64-level Merkle tree accumulator and about 3,8M constrains), taking 450 seconds to create the proof and consuming 30 GB of ram. For the proofs, we used Groth16 with Circom, witness generation in C++, and rapidsnark.

On the positive side, the proof that needs to be computed to generate a fraud proof is small enough (50k) that it can easily be generated in a browser. This allows users to challenge a voting batch without downloading any extra specialized software.

Future research

They are some areas that we want to explore further:

- Verify standard keccak/ecdsa/sec256k1 signatures via snarks. We believe that soon will be able to verify these schemes, which will open two possibilities:
- To provide a proof that the BabyJub key has been derived from a secp256k1 key. This only needs to be done once.
- To verify the vote signature itself.
- To provide a proof that the BabyJub key has been derived from a secp256k1 key. This only needs to be done once.
- To verify the vote signature itself.
- Verify storage proofs inside a snark. We think that this kind of complex circuit could be easily integrated via a zkVM, though the cost could be really huge. We are also worried about Ethereum clients sunsetting archive nodes to prioritize higher gas limits, so another point of research is to try to use methods other than EIP1186 for storage proofs.
- For the computation of the tally, embed some kind of opcodes to be executed inside a zkVM, allowing generic programmable voting circuits.
- Generate a voting proof in the browser, mix via batching, and recursively aggregate the results, similarly to the zk.money protocol. This would result in increased privacy for the voting process.
- Allow snarks to be computed at the browser level in a distributed way, even if they are huge to compute. This means not relying on big, easy-to-target servers and instead being fully P2P, thereby giving all the power to voters.
- Embed privacy and mixing in the voting protocol at a network-level
- Find a cryptoeconomics model that makes sense and is fully interoperable with eth2.0
- As a more general goal, generate a unique proof that can easily be verified. This opens the possibility for any programmable L1 and L2 (EVM or not) to react to any Ethereum voting results. The long-term goal is to be able to vote on any chain and to verify the results on any another chain. This could become some kind of gold-standard for cross-rollup/chain storage proofs verification via snarks.