OVOTE (v0.5) : Off-chain Voting with On-chain Trustless Execution

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

[1]Aragon ZK Research
[2]Vocdoni
(both [1] and [2] are part of the Aragon ecosystem
and are being funded by Aragon Association)

June 2022

This is a draft. Current document version: 2022-06-21.

Abstract

We present OVOTE, a Layer 2 design for voting on Ethereum using validity-proofs (zkSNARK proofs), which are similar to zkRollups. The census is constructed based on public keys of the voters. Votes are aggregated off-chain and proved on-chain through a zkSNARK proof. This results in constant gas costs while scaling up to thousands of voters via a single Ethereum transaction. It is hoped that this solution will help address the issue of low engagement in DAOs on Ethereum, which is partly due to the cost of on-chain voting.

This document provides a technical overview of OVOTE v0.5, which is an already usable implementation that is consisting of the following modules:

- Circuits: zkSNARK (Groth16) Circom circuit, for off-chain computation of the aggregation of valid user votes and result computation.
- Smart Contracts: on-chain registry of processes, also verifies the validity proofs and results and triggers on-chain tx execution.
- Node: OVOTE node, similar to a zkRollup node. Aggregates the votes and generates the zkSNARK proof.
- Client lib: typescript library used in the user's browser, to create keys, signatures and cast votes.

We also explore further developments of OVOTE, like the inclusion of anonymous voting, together with longer term challenges which involve significant research efforts such as exploiting the potential of SNARKs recursion.

Target audience: web3 developers, advanced DAO practitioners
Keywords: SNARKs, zero-knowledge, Ethereum, voting, DAOs

Contents

1 Introduction **3**

2 Properties of OVOTE v0.5 **4**
- 2.1 Properties . . . 4
- 2.2 Missing properties . . . 4
- 2.3 Limitations . . . 5

3 Cryptographic primitives **5**
- 3.1 zkSNARK . . . 5
- 3.2 Hash function . . . 5
- 3.3 Merkle Tree . . . 5
- 3.4 Elliptic curve . . . 6

4 System description **6**
- 4.1 Actors . . . 6
- 4.2 System flow . . . 6
- 4.3 Modes of operation and Smart Contract timelines . . . 7
  - 4.3.1 Multisig mode . . . 7
  - 4.3.2 Referendum mode . . . 9
- 4.4 zk circuit . . . 10
- 4.5 Census creation . . . 11
  - 4.5.1 Keys creation . . . 11
  - 4.5.2 Census Tree . . . 11
  - 4.5.3 Census proof . . . 12
- 4.6 Process creation . . . 12
- 4.7 Votes aggregation and results computation . . . 13
- 4.8 Results publication . . . 13
  - 4.8.1 Multiple nodes . . . 13
  - 4.8.2 Vote receipt . . . 14

5 Implementation **14**

6 Future work **15**
- 6.1 Short term concrete iterations . . . 15
  - 6.1.1 On-chain anonymous voting . . . 15
  - 6.1.2 OVOTE DAO . . . 16

Acknowledgements **16**

References **17**

1 Introduction

Many DAOs on Ethereum face the problem of low engagement. This is in part due to the cost of participating in decisions by voting on-chain. While such costs are no longer as exorbitant as they were in early 2021, they are still not negligible. To address this problem, various solutions have been proposed to move the voting off-chain, but with the result of the voting being implemented on-chain.

OVOTE is such a solution. Aragon ZK Research [1] has developed a proof-of-concept system that allows for off-chain voting based on censuses made of public keys of the census members, with subsequent on-chain execution. We have leveraged, tried and tested technologies, which are currently being used in production by projects such as Polygon-Hermez and Tornado Cash.

This document provides an overview of OVOTE v0.5. It explains the technical choices made, provides details of its implementation and maps out possible future improvements. Thus, target audience of this document comprises primarily web3 developers and advanced practitioners. The code itself is already available on GitHub, under an open source license (GPL v3.0). Existing code from previous projects was reused whenever possible.

This is a project that we started 4 months ago in the Aragon ZK Research group, continuing the ideas that we have been discussing in Vocdoni [2] for the past year, with the aim to contribute to the Aragon [3] ecosystem. The focus of the work we present here is not on the theoretical design, but on a concrete design with a usable implementation. Although we are aware that it is far from ideal, we consider it a valuable initial step towards the direction of using zkSNARKs for voting, a technology which, in our opinion, has a huge potential in voting systems. Hence, we are already working on additional designs.

The rest of the document is organized as follows. In 2 we review the properties as well as limitation of the version of OVOTE we present here. In 3 we list the cryptographic primitives we are building on and justify why we selected each of them. 4 discusses how the system works internally. We describe the main actors, the system flow, the operation modes available, and the building blocks and their interaction. 5 is a short section pointing to the corresponding code repositories. Finally, in 6 we explore further developments of OVOTE, like the inclusion of anonymous voting, together with longer term challenges which involve significant research efforts such as exploiting the potential of SNARKs recursion.

2 Properties of OVOTE v0.5

This section describes the properties as well as the known limitation of the version of OVOTE we present here (v0.5).

2.1 Properties

The current version of OVOTE has the following properties:

Universal verifiability The way we use the validity proofs makes the results verfiable by any actor.

Unforgeability (tamper-evident) Neither users nor the system operators can manipulate the votes or add fake votes (eg. votes by users that are not in the census).

Trustless Thanks to the previous properties, the system can run in a trustless way, that is to say, no-one needs to trust in any specific party.

Binding execution Due to the universal verifiability property, the proof verification can trigger on-chain actions (e.g. moving funds of a DAO) in a trustless way, directly from the voting process result.

off-chain/gasless voting users vote off-chain, and the OVOTE-node aggregates the computation and verification of all the votes, signatures and census-proofs, in a succinct validity proof, which is sent to the Smart Contract. The only transactions executed on-chain are the *process creation* and the *results publishing*.

Scalability Thousands of votes can be aggregated in a single Ethereum tx. This is enables multisigs orders of magnitude larger than current on-chain solutions, among other benefits.

Chain agnostic census 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 an OVOTE census could be used in Ethereum mainnet, but also in other chains.

2.2 Missing properties

The following properties are missing in the version we present here, but we plan to addressed them in future versions (see Section 6).

No voter privacy While the relation between votes and public keys is not published in the blockchain, the OVOTE-node will know these relations and could make them public.

No token-based voting The scope of OVOTE is not about token-based voting (Section 6.1.2 discusses how we plan to introduce token-based voting).

2.3 Limitations

We identify the following as the three main limitations of OVOTE v0.5.

- User friction during census creation (this is solved by the schema proposed in Section 6.1.2)
  - 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.
- Not fully decentralized (partially solved by Section 4.8.1)
- Requires a trusted setup ceremony. This is due the usage of Groth16.

Note that this last limitation cannot be overcome without switching to a different proof system, which would require an almost complete refactoring.

3 Cryptographic primitives

This section describes the different cryptographic primitives used in OVOTE. The reason for the choices we made is twofold. First, there are some specific constraints linked to the EVM (Ethereum Virtual Machine), which only supports bn254 pairing. Second, we sought to leverage work previously done in Circom and Go.

3.1 zkSNARK

To generate zkSNARKS, we use the Circom 2.0 zero-knowledge circuit compiler [4] [5] with Groth16 [6] and the pairing-friendly bn254 elliptic curve. This constrains the other cryptographic primitives we use.

3.2 Hash function

Traditional hash functions are expensive in terms of the number of constrains. For this reason, we use *Poseidon* [7] [8] which is a *snark friendly* hash function. While the circuit of the Keccak256 Circom implementation [9] has around $150k$ constraints, the Poseidon Circom implementation [10] for 2 inputs has only around 250 constraints.

3.3 Merkle Tree

We use the binary sparse Merkle Tree from iden3 [11], in which a data entry $e = (k, v)$ is composed of a *key* k and a *value* v. The position in the tree is uniquely determined by the binary representation of its key k, specifically the hash of the key. More specifically, we use a custom implementation which parallelizes by CPUs for faster computation named *arbo* [12].

3.4 Elliptic curve

We need an elliptic curve that voters will use to sign their votes. Traditional curves are not snark-friendly, which would lead to a large amount of constraints in our circuit. For this reason, we use the *BabyJubJub* curve which is embedded in the bn254 pairing field. We can then compute the embedded elliptic curve operations in the native field of our constraints system over the pairing field. For the signature scheme, the EdDSA algorithm is therefore used. A detailed description of BabyJubJub can be found at EIP-2494 [13] and in iden3's BabyJubJub document [14].

According to the current standard approach for zkRollups, we derive the BabyJubJub private key from a user's secp256k1 signature. This can be done using Metamask, without needing to store the private key in the browser. The signature specification follows EIP-712 [15], and more details of this key generation approach can be found in Section 4.5.1.

4 System description

This section, which constitutes the core of this document, provides a detailed description of how OVOTE works internally.

4.1 Actors

We have defined the following actors.

Census members List of eligible voters.

Voters Census members who have already cast their vote of are in the process of doing so.

Organizer Performs two main functions i) build the census (*census creator*) and ii) set up the voting processes parameters (*process creator*).

Node Receives the votes, and generates the validity proof for the result.

4.2 System flow

As shown in Figure 1, there are four main phases: census creation, vote casting, votes aggregation, results publishing.

1. **Census and process creation**
   - 1.1 The organizer builds the census by placing the census members' public keys (K) and weights (w) in the Merkle Tree leaves.
   - 1.2 The organizer makes the Merkle Tree publicly available to the voters.
   - 1.3 The organizer publishes the Census root (R) in the Ethereum Smart Contract, which initiates the actual voting process.

2. **Vote casting**

   2.1 The user retrieves their census proof siblings (s_i) from the Census Tree (which proves that the user's public key (K_i) belongs to the census C).

   2.2 The user performs the signature (sig_i) over their vote (v_i), with their private key (k_i) which public key (K_i) belongs to the census C.

   2.3 The user sends to the Node their envelope $e_i = \{v_i, K_i, w_i, s_i, sig_i\}$.

3. **Votes aggregation**

   3.1 Once the defined *ResultsPublishingStartBlock* is reached, the Node generates the witness from all the users data.

   3.2 The Node generates the validity proof (π).

   3.3 The user retrieves π from the Node. This can be done by any user.

4. **Results publishing**

   4.1 User sends the result together with π to the Ethereum Smart Contract. This can be done by any user.

   4.2 Smart Contract verifies the result and π, and triggers the corresponding on-chain action.

4.3 Modes of operation and Smart Contract timelines

The current implementation of the zk circuit allows for two modes of operation: multisig and referendum. The main difference among the two is in the way the results are handled by the *Smart Contract* at the results publishing phase. In addition, as discussed in Section 6.1.1, we are working on an additional circuit that, based on much of the work we present here, will allow for a third operation mode: *on-chain anonymous voting*.

It is worth noting that a given census must not be uniquely related to a specific voting process, i.e. the same Census Root could be reused for many different voting processes. Furthermore, as discussed in Section 4.1, the Process Creator can be different than the Census creator. This would allow for use cases where an organization builds a census, and once everybody agrees on that census, anybody can create voting processes for that census, using its Census Root.

4.3.1 Multisig mode

In multisig operation mode, the voting process only involves one possible answer (*support*) that the census members implicitly choose when they cast their vote. In other words, in the multisig mode, census members can only choose between giving support, which is made by casting a vote, or abstaining. This is the most straightforward usage of the OVOTE and the operational timeline is very similar to the traditional multisig mechanism. As shown in Figure 2, it consists on aggregating the users votes as they come in and, as soon as the minimal threshold support is reached, the *result + proof* can be sent to the Smart

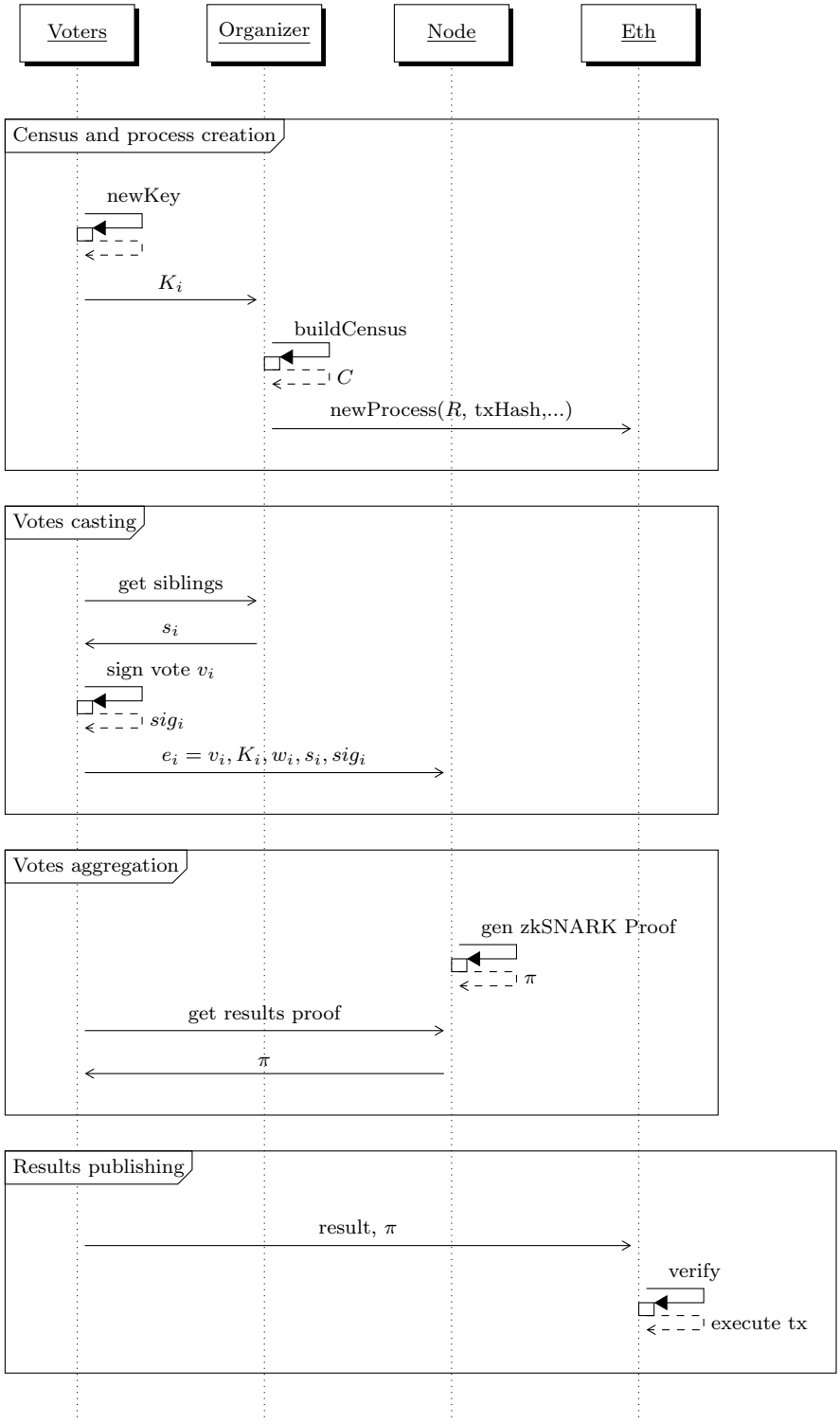

Figure 1: OVOTE system flow

Contract, which verifies it and, in case of a successful verification, triggers the pre-established actions, for instance a funds movement.

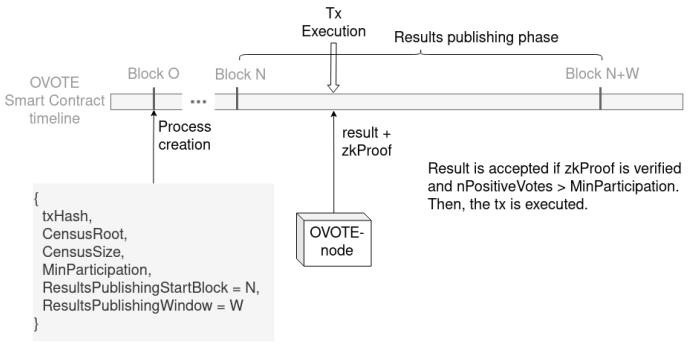

Figure 2: Timeline of a multisig mode process in the Smart Contract.

4.3.2 Referendum mode

In the referendum mode, the voters are given two options to choose from: *yes/support* and *no/against*. Thus, compared to the multisig mode, the referendum mode enables the expression of explicit opposition. Figure 3 shows how this is implemented. When creating the new process, the *Organizer* specifies not only the *ResultsPublishingStartBlock*, but also a *ResultsPublishingWindow*, which indicates the number of blocks after *ResultsPublishingStartBlock* when the Smart Contract will be accepting new *results*.

The Smart Contract accepts new *results* if the *proof* verifies correctly, and if the zk circuit public input *nVotes* (see Section 4.4) used in that result is larger than the previous result published. This partially mitigates the censorship capacity of the OVOTE-nodes to intentionally exclude specific votes. User can send their vote to multiple nodes, and the node which present the results and proof including the most votes is the one that will prevail.

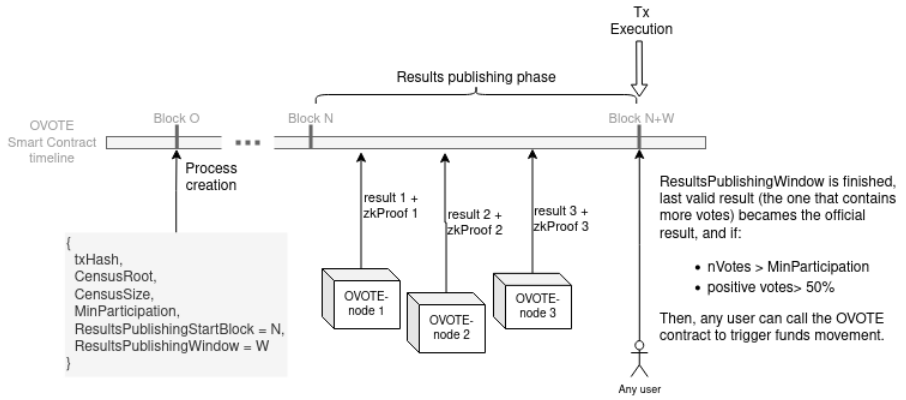

Figure 3: Timeline of a referendum process in the Smart Contract.

4.4 zk circuit

The inputs of the zk circuit are gathered in Table 1. This circuit defines a set of constraints that must be satisfied in order to accept a given result in the *Smart Contract*.

Public inputs		Private inputs	
Input	Description	Input	Description
chainID	hardcoded in the Contract deployment	v_i	vote
processID	determined by process creation	i	index
censusRoot	determined by process creation	K_i	public key
receiptsRoot	determined by results publication	w_i	weight
nVotes	determined by results publication in the Contract	sig_i	signature
result	input from the Contract call	s_i	siblings
withReceipts	input from the Contract call, can be 0 or 1, indicates if the process uses receipts	t_i	receiptsSiblings

Table 1: zk circuit public and private inputs

The checks defined by the circuit constrains are:

- Each used K_i exists in C for the public R
- Each vote v_i is signed by one of the valid K_i
- Each signature sig_i is valid
- A K_i 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 K_i weight w_i
- All the votes are cast for a specific $chain_{ID}$ and $process_{ID}$ (to prevent proof re-usability attacks)
- If enabled, the *receipts* are well computed (more details at Section 4.8.2)

To prevent double spending, that is, prevent the same K_i from casting more than one vote we sort $\{i, v_i, K_i, w_i, sig_i, s_i, t_i\}$ by i and the circuit checks that $i - 1 < i$, that is, ensuring that there are no repeated indexes, and thus, tahat there are not repeated public keys. This approach takes 64 constraints for each vote.

Alternatively we could use a *nullifier* approach, where each user together with the vote v_i, s_i, sig_i, w_i, where the nullifier would be the hash of $\{sk, process_{ID}\}$. However, this approach takes around 261 constraints for each nullifier that we need to check assigned to each vote.

With the sorting solution, we get a reduction of $\sim 4x$. For instance, in a census of 10.000 users, the nullifier approach would take 2.61M constraints, while the index approach requires only 0.64M constraints.

4.5 Census creation

4.5.1 Keys creation

Users can directly generate a BabyJubJub EdDSA key pair and store the private key in memory, but we aim to follow the same approach that current zkRollups do for key generation for more user-friendliness.

We assume that users already have an Ethereum key pair, then, each user generates their private key following the EIP-712 [15], signing with their Ethereum *private key*. From that signature (additionally it could be hashed), users generate their key pair on the BabyJubJub elliptic curve, following the EdDSA scheme, obtaining their public key K, such that $K = k \cdot G$, where $k = sign_{eth}(key\_generation\_bytes)$ and G is the BabyJubJub group generator.

This is the approach used in Hermez 1.0 [16], preventing users from needing to manage the BabyJubJub keys, as they can derive it from their already existing Ethereum keys. It is important to note that the generated Ethereum signature should not be leaked, as it is used as private key (k).

4.5.2 Census Tree

We follow the Merkle Tree design described in 3.3. Each user sends K to the Census creator. The Census creator can be the voting process organizer. The Census is the set of Ks that will be able to vote. We denote the Census by the set $C = \{i, K_i, w_i\}$, where w_i defines each K_i assigned *weight* (the voting power of each K_i), and the index is an incremental integer used to determine the position of each leaf in the *Census Tree*.

The Census creator will add all the user's K_i to a Merkle Tree (from now on, the *Census Tree*), together with its assigned w_i and *index* i. The Census Tree defines C.

As shown in Figure 4, in the Census Tree leaves, the index is placed at the *key* of the leaf, and K coordinates and w are hashed using Poseidon as $H(K_i.X, K_i.Y, w_i)$. The hash output is placed in the value of the leaf.

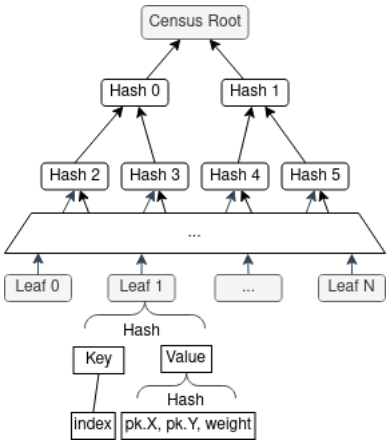

Figure 4: Census tree as a Merkle tree.

As the Census Tree is generated from a list of K_i and w_i, 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 Smart Contract that builds the Census Tree out of all K_i of token depositors, assigning to each K_i it's corresponding w_i based on the amount deposited. We're leaving this out of this initial iteration of OVOTE, but a bit of more detail on this idea can be found at 6.

4.5.3 Census proof

The Census Proof (p) is a Merkle Proof, which is a *proof of membership* of a leaf in the tree under a given root R. This proof is formed by the *siblings* in the tree across the path from the leaf to the root, together with the index, the public key and it's weight: $p_i = \{s_i, i, K_i, w_i\}$.

The same kind of proof is used for the *Vote Receipts* (4.8.2).

4.6 Process creation

Once the census is completed, the R is used as the *Census Root*, which is sent by the Organizer to the Smart Contract, to indicate the creation of the voting process. Together with R, the organizer also sends the following data to the Smart Contract:

- *Census size* ($|C|$): the number of Ks contained in C.
- *Results publishing start block*: the block number where the Smart Contract 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.

4.7 Votes aggregation and results computation

In order to vote, users generate a BabyJubJub EdDSA signature of the vote value (v_i), together with the $chain_{ID}$ and $process_{ID}$ to prevent vote re-usability across different processes and chains. User then obtains

$$sig_i = sign_k(chain_{ID}, process_{ID}, v_i)$$

Users send their vote envelope $e_i = \{v_i, p_i, sig_i\}$ 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 zkSNARK proof.

The node prepares the *witness* from the votes data and generates the zkSNARK proof (π), fullfilling the OVOTE circuit.

4.8 Results publication

Once the proof π is computed, it is made public by the Node. This design is to prevent the Node from sending transactions to Ethereum, which would require a wallet with all the security consequences derived from it.

Any user can retrieve the proof from the Node, and then send it to the Smart Contract together with the claimed result r. The DAOs using OVOTE can have a mechanism where the DAO pays back to the user the gas costs of such transactions. Another option would be to use Ethereum gas stations [17].

For the current OVOTE Contracts, the zkSNARK proof verification takes 262k gas, and the results publication (which includes the zkSNARK proof verification) takes between 402k and 990k gas, depending on the chosen approach. These numbers are estimations, the Smart Contract can be optimized to lower these numbers.

4.8.1 Multiple nodes

Once the *ResultsPublishingStartBlock* is reached, any user can send the result r together with the proof π to the Smart Contract. A potential problem is that a Node might compute a result that includes only a subset of all the votes, based on the Node operator's preferences, in order to reach a different result from the one that would be obtained without excluding some votes.

To prevent this in *referendum mode* (4.3.2), the OVOTE Smart Contract accepts multiple result/proof pairs $\{r, \pi\}$ during a certain period of time (the *ResultsPublishingWindow*). New pairs are accepted as long as new result corresponds to a larger number of votes than the previous one. This would allow the users to send their votes to multiple OVOTE-nodes. Then, if some Node discards their votes, another Nodes will have larger number of votes included, and its result will prevail.

The fact that in the *multisig mode* (4.3.1) just a minimal threshold is needed makes the solution against censorship much simpler. 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. Moreover, as last resort, a new Node could be deployed by people suspecting censorship by other Nodes.

4.8.2 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 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 receiptness, it can be disabled.

The way how vote receipts work is the following. When the Node aggregates the votes, also it builds the set $T = \{i, K_i, w_i\}$, which is a Merkle Tree (from now on *Receipts Tree*) that contains all the i, K_i, w_i of the all votes that are being aggregated in the proof.

The *Merkle proof* of inclusion of the set of T is denoted as $t_i = \{s_i, i, K_i, w_i\}$ (s_i being the siblings of each Merkle Proof), and is used as a private input in the circuit, and the *Receipts Root* R^t (the Merkle Root of the Receipts Tree) is used as a public input. In this way, the results publication in the Smart Contract, includes R^t, and the users can check that their vote has been included as their K_i is in the T under the published R^t, proven by t_i.

The circuit ensures that all Ks used for the receipts are the Ks that are being used also for the votes computation, thus guarantees that $T \subseteq C$.

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 it 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*).

5 Implementation

The OVOTE implementation consists of four modules. These are available on GitHub under the GNU General Public License v3.0 and AGPL-3.0.

- Circuits: zkSNARK (Groth16) Circom circuit, which proves the correct off-chain computation of the aggregation of valid user votes and result computation. GitHub [18]
- Smart Contracts: on-chain registry of processes, also verifies the validity proofs and results and triggers on-chain tx execution. GitHub [19]
- Node: OVOTE node, similar to a zkRollup node. Aggregates the votes and generates the zkSNARK proof. GitHub [20]
- Client lib: typescript library used in the user's browser, to create keys, signatures and cast the votes. GitHub [21]

Instructions on how to integrate OVOTE with DAOs will be published on the GitHub repositories later.

6 Future work

As we have seen, this document describes a system which we see as a first step in the direction of using *validity proofs* for off-chain votes computation with on-chain verification. That is why we have already started to research new features that could be added in further OVOTE iterations, or even for completely new unrelated designs.

One of the more straightforward features would be adding homomorphic encryption to the votes. The necessary checks would be done over the encrypted votes inside the circuit, so the content of the votes would not be known until the publication of decryption keys 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.

Furthermore, another research line is the use of ring signatures inside zkSNARK circuit, which would allow users proving that they belong to the census (or a subset of the census) without revealing who they are, and later all the signatures verification would be proven inside a zkSNARK circuit by the node. This would bring anonymous voting while having the scalability that we have with OVOTE.

Some other interesting lines of research are being able to verify *EthStorageProofs* (Ethereum state Patricia Merkle tree proofs), to verify inside the circuit that the voter is an Ethereum address that holds a certain amount 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 large 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 *zkSNARK proof* inside another *zkSNARK 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 *proofs*, which is the one that would be verified in the *Smart Contract*. This would enable privacy voting.

6.1 Short term concrete iterations

While the previous described ideas need more research, in this sub-section we describe some short-term iterations that we have in mind to implement right after the OVOTE first iteration.

6.1.1 On-chain anonymous voting

As explained earlier, ideally we would have a 1 level of recursion *rollup*, similar to OVOTE, but with users sending a zk-Proof proving that they are in the census and the node aggregates all the proofs in a proof, having then a system that allows off-chain trustless voting with anonymity. This is in our plans, but

there is a immediate way to have anonymous on-chain voting with the current OVOTE circuit design.

The idea of having *on-chain anonymous voting* would be reusing the OVOTE created censuses and part the OVOTE circuit, and making users vote by sending to the on-chain Smart Contract a zk-Proof (generated in the browser), which proves that they belong to the Census, without revealing which key. And if the proof is correct their vote is computed by the Smart Contract. In this way, users can vote in an anonymous way without revealing who they are in the census, but proving that they belong to the census.

In a simplified way, it would be a mix between the OVOTE project and what we implemented in Vochain's anonymous voting [22], but directly verifying the zk-Proofs on Ethereum chain.

This could be done reusing an already existing census of the OVOTE, with the main difference that would be anonymous and users paying gas, and, by consequence, it might be expensive (economically) for users to sent their votes and zk proofs to Ethereum main net. For high stake use cases this can be affordable, but most of use cases might be more feasible by using this approach in a EVM-chain with less congestion such as Polygon or GnosisChain.

However, it is important to keep in mind that this *on-chain Anonymous Voting* is a consequence of the *OVOTE* circuit design, but not a project in itself. There are already existing projects covering on-chain anonymous voting with specific designs, such as MACI [23].

6.1.2 OVOTE DAO

An interesting idea to build on top of the OVOTE, is *OVOTE-DAO*. The main idea 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 Smart 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 Smart 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 OVOTE. This could be managed by a Smart Contract on top of the OVOTE Smart Contract.

Summarizing, this would be an 'off-chain DAO with on-chain funds movement through validity-proofs', kind of a 'DAO-zkRollup', where users make token deposits to the DAO Smart Contract, and then they vote off-chain based on the weight of their deposited tokens, and a validity proof is published to the DAO Smart Contract, which if correct, proves the correctness of the results and triggers a transaction execution (which, can be DAO funds movement).

Acknowledgements

We would like to thank Vincenzo Iovino for useful feedback. We are also indebted to Barry Whitehat and Jordi Baylina, as many of the ideas that we are using come from the rollup concept. We also want to thank Adrià Massanet,

with whom during the last year we have been iterating ideas for scalable voting solutions using zkSNARKs.

Aragon ZK Research and Vocdoni are projects currently funded by Aragon Association [24].

References

[1] Aragon ZK Research website. https://research.aragon.org.

[2] Vocdoni website. https://vocdoni.io.

[3] Aragon website. https://aragon.org.

[4] Jose L. Muñoz-Tapia, Marta Belles, Miguel Isabel, Albert Rubio, and Jordi Baylina. Circom: Robust and scalable language for building complex zero-knowledge circuits, 2022. https://www.techrxiv.org/articles/preprint/CIRCOM_A_Robust_and_Scalable_Language_for_Building_Complex_Zero-Knowledge_Circuits/19374986/1/files/34409498.pdf.

[5] Circom code. https://github.com/iden3/circom.

[6] Jens Groth. On the size of pairing-based non-interactive arguments. Cryptology ePrint Archive, Paper 2016/260, 2016. https://eprint.iacr.org/2016/260.

[7] 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.

[8] Poseidon hash website. https://www.poseidon-hash.info.

[9] Keccak256 Circom code. https://github.com/vocdoni/keccak256-circom.

[10] Poseidon circom code. https://github.com/iden3/circomlib/blob/master/circuits/poseidon.circom.

[11] Jordi Baylina and Marta Bellés. iden3's sparse merkle trees. https://docs.iden3.io/publications/pdfs/Merkle-Tree.pdf.

[12] arbo. https://github.com/vocdoni/arbo.

[13] EIP-2494. https://eips.ethereum.org/EIPS/eip-2494.

[14] Barry WhiteHat, Jordi Baylina, and Marta Bellés. iden3's babyjubjub elliptic curve. https://iden3-docs.readthedocs.io/en/latest/_downloads/33717d75ab84e11313cc0d8a090b636f/Baby-Jubjub.pdf.

[15] EIP-712. https://eips.ethereum.org/EIPS/eip-712.

[16] Hermez 1.0. https://github.com/hermeznetwork.

[17] Ethereum gas station network (GSN) website. https://docs.opengsn.org/.

[18] OVOTE zk circuits code. https://github.com/aragonzkresearch/ovote/tree/main/circuits.

[19] OVOTE smart contracts code. https://github.com/aragonzkresearch/ovote/tree/main/contracts.

[20] OVOTE node code. https://github.com/aragonzkresearch/ovote-node.

[21] OVOTE client library code. https://github.com/aragonzkresearch/ovote/tree/main/clientlib.

[22] Vochain anonymous voting. https://docs.vocdoni.io/architecture/protocol/anonymous-voting/zk-census-proof.html.

[23] MACI documentation. https://privacy-scaling-explorations.github.io/maci/.

[24] Aragon Association website. https://aragon.org/aragon-association.