DAVINCI

Authors: Pau Escrich and Jordi Piñana from Vocdoni Association

Last update: June 5, 2025

Abstract. DAVINCI is the evolution of the Vocdoni voting protocol, designed to empower civil society by providing essential tools for secure, verifiable, and anonymous digital voting. Leveraging recent advancements in zero-knowledge (ZK) proof systems and blockchain technology, DAVINCI transitions to a specialized ZK-rollup system that inherits network security from settlement layers like Ethereum mainnet. The system relies on cryptographic proofs to ensure integrity and security, eliminating the need for centralized authorities. By integrating ZK-SNARKs and threshold homomorphic encryption, DAVINCI enables end-to-end verifiability, privacy, and trustlessness in the voting process. The protocol employs a distributed key generation among sequencers, coordinated via Ethereum smart contracts, and utilizes Ethereum data blobs for data availability. With a focus on accessibility, scalability, receipt-freeness, and automation, DAVINCI aims to facilitate high-frequency, low-cost voting, fostering mass adoption of e-voting and simplifying civil participation. Finally, the introduction of the Vocdoni token (VOC) aligns incentives among participants, ensuring the system's sustainability and enabling decentralized governance.
Moreover, the design of DAVINCI is grounded in practicality; all components have been implemented using current technologies and have undergone proof-of-concept testing. This ensures that the proposed architecture is not merely theoretical but a viable solution ready for short-term deployment.

Marta: Change Vocdoni to DAVINCI.

Marta: Mention that the spec can be tailored to the needs of the specific vote. So, here we present some default configuration, but some of the pieces can be changed (specially the cryptographic schemes or curves). For this reason, we tried to make this document modular (like the code), to facilitate the changes and the analysis of the consequences.

Marta: Separate the roles of the sequencers from key generators?

Table of Contents

Vocdoni

Vision and background. Voĉdoni, meaning "to give voice" in Esperanto, embodies our mission to empower civil society from the grassroots level. We aim to build essential primitives and tools that enable any collective—from small groups to millions of citizens—to be heard, regardless of their circumstances or available resources. Our philosophy envisions voting beyond traditional nation-state elections; we see it as a collective signaling mechanism with cryptographic guarantees of integrity and outcome. To address the challenge, we developed a end-to-end verifiable and anonymous voting system designed to work on any device, including smartphones. We also successfuly deployed an infrastructure that maximizes resilience, neutrality, and transparency.

In 2018, when Vocdoni began, zkSNARKs was just an emerging technology. We chose to base our solution on a customized Byzantine fault tolerant layer-1 (L1) blockchain called Vochain. This allowed us to achieve scalability (approximately 700 transactions per second), to leverage advanced cryptographic tools that are prohibitively expensive on EVM-based blockchains, and enable users to send voting transactions without costs. This experience has provided us with invaluable insights that have enabled us to overcome technical and operational challenges. Although this solution has effectively met user requirements and demonstrated its viability, further development is essential for its widespread adoption as a universal voting protocol.

DAVINCI represents the evolution of the Vocdoni voting protocol. It integrates smart contracts for orchestration, a zkSNARK-based state machine for verifying and accumulating votes, and a decentralized data availability layer to ensure censorship resistance.

Design principles. To build the DAVINCI stack, we adhere to the following design principles:

1. **Cryptography as the source of truth**: we rely exclusively on cryptographic proofs to ensure the integrity and security of the voting process. By trusting only in cryptography, we eliminate the need for centralized authorities, making the system inherently secure and transparent.
2. **Trustlessness**: our system operates without requiring trust in any single party. Through cryptographic protocols and decentralized infrastructure, we ensure system integrity and prevent compromise from any malicious actor.
3. **End-to-end verifiability**: every voter can verify their ballot from casting to result computation (individual verifiability). Additionally, any third party can audit the election data to confirm results (universal verifiability) and verify that each vote comes from a uniquely registered voter (eligibility verifiability). Transparent cryptographic mechanisms make this possible.

4. **Composability**: the system is modular, consisting of interchangeable components that can be rearranged or integrated with external systems via adaptable interfaces. This allows for redundancy, flexibility, and seamless integration with third-party applications, exemplified by our voting-as-a-service APIs.
5. **Accessibility**: Vocdoni's voting platform (App) is open source, universally available and user-friendly. The interface is intuitive for all users, including those less familiar with technology, and accommodates voters that use assistive technologies like screen readers.
6. **Open source**: by releasing our code openly, we invite anyone to audit and contribute, enhancing security and fostering community engagement. Transparency prevents security through obscurity and accelerates innovation.
7. **Resilience**: we design for robustness against hardware failures, network outages, and censorship. Infrastructure decentralization and distributed ownership enhance system availability and resistance to attacks.
8. **Scalability**: our solution processes votes at high throughput, exceeding current requirements to accommodate future growth. The decentralized infrastructure scales organically as usage increases, ensuring consistent performance.
9. **Receipt-freeness**: to mitigate risks of collusion, coercion, and vote-buying, the system enables voters to verify their votes without being able to prove to others how they voted, thus reducing incentives for coercion and bribery. Additionally, voters are allowed to overwrite their votes while maintaining secrecy.
10. **Automation**: we minimize human intervention through smart contracts and cryptographic protocols, reducing costs and human error. Automation ensures consistent operation and frees resources for voter support and auditing.

Components overview. Below, we detail the components of the architecture that collectively support the operation and management of the voting system.

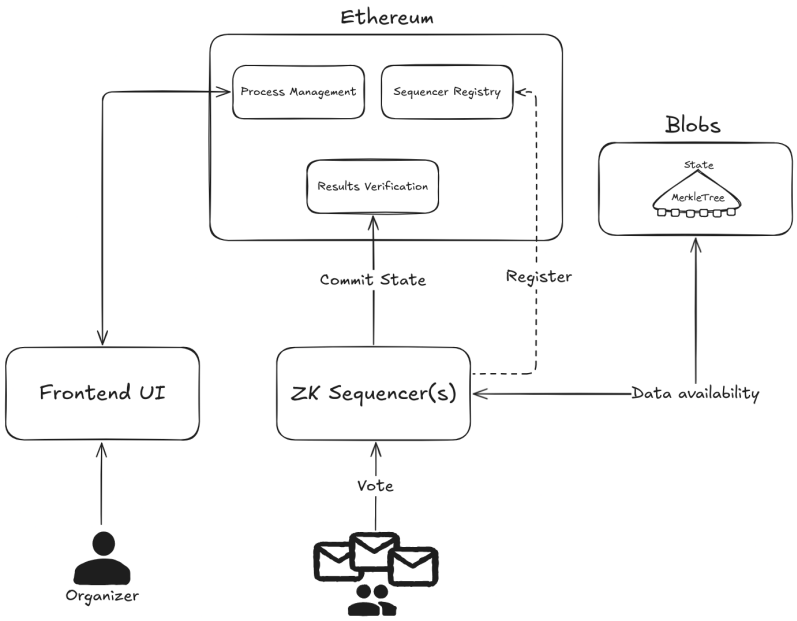


Fig. 1. Caption.

1. **Ethereum**. An Ethereum compatible network is used as the source of truth for the voting system. By leveraging an EVM blockchain, the process management ensures that all transitions are immutable and verifiable by all participants. To this end, we implement several smart contracts.
   - **Process management**: Responsible for the lifecycle management of voting processes. It includes the initiation, monitoring, execution, and closure of voting events.

- **Results verification**: For each voting process, this smart contract maintains the integrity of the cast votes and process lifecycle. It verifies that each state transition committed by a ZK sequencer adheres to the predefined rules.
- **Sequencer registry**: This smart contract keeps track of the existing available sequencers, stores the collateral to ensure good behavior, and it's used to coordinate the distributed key generation when a new Process is created.

2. **Sequencer**. The Sequencer is a specialized component designed to handle the voting process using zero-knowledge proof mechanisms. It ensures that all transactions related to this process are validated and sequenced. The Sequencers periodically commit the state of the voting process to Ethereum.
3. **Frontend user interface**. The user interface serves as the primary interaction layer for voters and organizers. It provides tools and functionalities needed by organizers to set up, manage, and oversee elections. This interface simplifies the complexities involved in managing a decentralized vote.
4. **Data availability**. For each voting process, the system needs to keep track of the current State Merkle Tree, which contains all information of such processes. A public data availability layer, ensures that all state transitions are available and verifiable and allows the participation of multiple sequencers within the same voting process.

1 Introduction

1.1 Contributions

- The Vocdoni voting protocol.
- The Vocdoni ballot protocol.

1.2 Related work

- State of the art of e-voting: `https://azkr.org/blog/evoting-review/`.

1.3 Paper organization

- Section 2: background.
- Section 3: high overview of the election process. We omit technical details.
- Section 4: cryptographic primitives used in the voting process. A description of the primitives but also the details of their instantiation, that is, the specific protocols being used.
- Section 5: description of the voting process.
- Section 6: a proposed protocol for ballots. The rules of this protocol are implemented in the process management smart contracts.
- Section 7: the Vocdoni token.
- Section 8: analysis of the proposed protocols. It includes a security discussion of the properties of the protocol, implementation details, and a performance evaluation.
- Section 9: we finish with some last remarks.

2 Background

2.1 Interplanetary file system (IPFS)

- Brief introduction to IPFS.

2.2 Ethereum

- Ethereum blockchain (say transactions have a cost fee).
- Ethereum smart contracts.
  Old text: Ethereum smart contracts are employed to orchestrate the voting process, manage state transitions, and store critical data, such as the current state root and encryption public keys. These smart contracts provide a trustless environment, ensuring that all participants can be confident that the voting process rules are correctly enforced.
- Ethereum data blobs.
  Old text: EIP-4844 is used for data availability, enabling the storage of state transition data in the form of Ethereum data blobs. This mechanism allows sequencers to collaborate on the construction and verification of the voting process state, ensuring efficient and decentralized data availability.

2.3 Zero-knowledge proofs

- Introduce ZK-SNARKs.
- Introduce arithmetic circuits (arithmetic circuit satisfiability).

3 Protocol intuition

All the voting process is governed by the rules encoded in three smart contracts: the *process management*, the *sequencer registry*, and the *results verification* smart contract. All these three contracts are deployed on the Ethereum blockchain, and ensure that the execution of all steps is transparent and unalterable. The protocol consists of 5 phases, as illustrated in Figure 2, and described below.

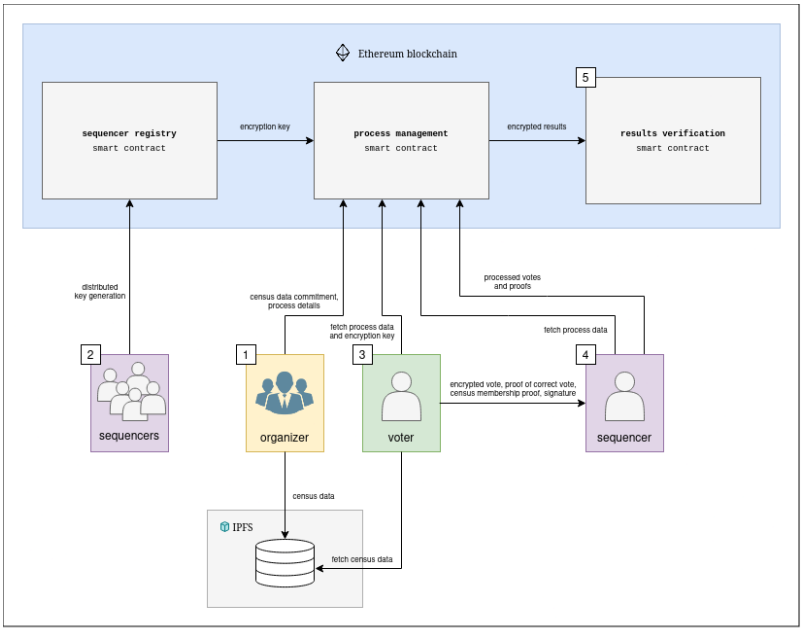


Fig. 2. Overview of the protocol flow.

Marta: TODO: rename the phases as they will appear in Section 5.

1 *Voting setup.* In the first phase, the organizing entity gathers all census data and generates a cryptographic commitment to this data. Additionally, the organizer defines the voting parameters, including the number of voting options and the vote-counting mechanism (e.g., weighted voting, quadratic voting, etc.). Once these details are set, the organizer submits a transaction to the process management smart contract on the Ethereum blockchain. This transaction publicly records the voting parameters and census commitment, ensuring transparency and preventing any subsequent alterations to the voting setup.

2 *Encryption key generation.* A designated group of participants, referred to as *sequencers*, collaboratively generate a shared encryption public key, which voters will use to encrypt their votes. This key is established through a distributed key generation (DKG) protocol, ensuring that no single party can control or reconstruct the corresponding private key independently. Sequencers have a dual role in the protocol. In addition to generating the encryption key, they are also responsible for collecting votes from voters during the election. Consequently, their public key and the necessary information to contact them off-chain must be registered in the sequencer registry smart contract. Once the encryption public key is securely computed and published in the sequencer registry smart contract, the voting phase can start.

3 *Voting.* Voters select their preferred option according to the rules established by the organizer and captured in the process management smart contract. Instead of submitting their votes directly on-chain, they send them off-chain to a sequencer of their choice for processing. To ensure privacy, votes are encrypted using the available encryption public key. Alongside the encrypted vote, each voter must also provide the following data: a proof of valid voting, demonstrating that the vote complies with the election rules; a proof of eligibility, verifying that they are registered in the census; and a proof of identity ownership, in the form of a digital signature, confirming that they are the legitimate voter and are not impersonating someone else.

Marta: In the above paragraph, we could include a short description about double-voting/overwriting: add the goal/idea of the commitment and the nullifier.

4 *Votes collection.* During this phase, sequencers collect encrypted votes from multiple voters along with their corresponding proofs, and verify the validity of these submissions. That is, sequencers verify the signatures, to ensure the votes were cast by legitimate voters, they verify the proof of compliance, confirming that each vote adheres to the polling rules, and the census membership proofs against the commitment to the census data that was originally registered in the process management smart contract. Once the sequencers have processed and verified all votes, they must prove that these verifications were performed correctly. Instead of submitting individual verifications for each vote, they generate a single zero-knowledge (ZK) proof that attests to the correctness of all verifications. Additionally, the sequencers reencrypt the votes and generate a proof of the correct reencryption computation. While this process does not alter the final tally, it prevents voters from decrypting their original vote. This step mitigates vote selling or coercion, as voters are no longer able to prove their choice to third parties. Finally, the sequencers submit the reencrypted votes along with their verification and reencryption proofs to the process management smart contract. The smart contract verifies all the proofs provided by the sequencers to check that they complied with the protocol.

5 *Results verification.*

Marta: I did not look into the results decryption/verification part yet. A short description should be given here.

Old text: At the conclusion of the voting period, the smart contract ceases to accept new state updates, effectively finalizing the process. The final State is then available on-chain for verification, providing an immutable record of the voting outcome.

4 Cryptographic primitives

In this section, we detail the cryptographic primitives used in Section 5. We briefly introduce elliptic curves, hash functions, Merkle trees, encryption schemes, and proof systems used in the protocol. We also specify at each step the concrete parameters with which each of the primitives is instantiated.

Marta: Maybe it should go under the protocol section?

4.1 Elliptic curves

DAVINCI relies on multiple elliptic curves to ensure interoperability with Ethereum, compatibility with available cryptographic primitives, and efficient in-circuit operations. On the one hand, SECP256K1 [5] is used because Ethereum public keys are elements of this curve. As DAVINCI assumes each voter holds a standard Ethereum address, cryptographic operations such as digital signatures and identity verification rely on SECP256K1 to match the Ethereum ecosystem. On the other hand, BN254 [8] is chosen because it is the curve supported by Ethereum's precompiled contracts for zero-knowledge proof verification, which makes it ideal for verifying SNARK proofs on-chain with minimal gas cost. then, BabyJubjub [2] is used as an inner curve for elliptic curve operations within arithmetic circuits. Finally, BLS12-377 [4] and BW6-761 [6] are used to enable recursive proof composition. More specifically, BLS12-377 acts as the inner curve for constructing proofs, while BW6-761 serves as the outer curve that verifies those proofs within larger zkSNARK circuits. This pairing enables succinct verification and composability of zkSNARKs within other zkSNARKs, which we use for DAVINCI 's aggregation and state transition logic. Below, we give the details of these curves.

Parameters.

- $p =$ `0xfffffffffffffffffffffffffffffffffffffffffffffffffffffffefffffc2f` (256-bit prime).
- $q =$ `0xfffffffffffffffffffffffffffffffebaaedce6af48a03bbfd25e8cd0364141` (256-bit prime).
- $r =$ `0x2523648240000001ba344d8000000086121000000000013a700000000000013` (254-bit prime).
- $s =$ `0x2523648240000001ba344d8000000007ff9f800000000010a10000000000000d` (254-bit prime).
- $t =$ `0x60c89ce5c263405370a08b6d0302b0bab3eedb83920ee0a677297dc392126f1` (251-bit prime).
- $u =$ `0x122e824fb83ce0ad187c94004faff3eb926186a81d14688528275ef8087be41707ba638e584e9190 3cebaff25b423048689c8ed12f9fd9071dcd3dc73ebff2e98a116c25667a8f8160cf8aeeaf0a437e69 13e6870000082f49d00000000008b` (761-bit prime).
- $v =$ `0x01ae3a4617c510eac63b05c06ca1493b1a22d9f300f5138f1ef3622fba094800170b5d4430000000 8508c00000000001` (377-bit prime).
- $w =$ `0x12ab655e9a2ca55660b44d1e5c37b00159aa76fed00000010a11800000000001` (253-bit prime).

Elliptic curve groups.

- SECP256K1 curve: $E^{\mathsf{SEC}}/\mathbb{F}_p$ defined by equation $E^{\mathsf{SEC}} : y^2 = x^3 + 7$, with group $\mathbb{G}^{\mathsf{SEC}}$ of prime order q.
- BN254 curve: $E^{\mathsf{BN}}/\mathbb{F}_r$ defined by equation $E^{\mathsf{BN}} : y^2 = x^3 + 3$, with subgroups $\mathbb{G}_1^{\mathsf{BN}}, \mathbb{G}_2^{\mathsf{BN}}$ of prime order s, and an efficiently computable pairing $e : \mathbb{G}_1^{\mathsf{BN}} \times \mathbb{G}_2^{\mathsf{BN}} \to \mathbb{G}_T^{\mathsf{BN}}$, where $\mathbb{G}_T^{\mathsf{BN}} \subset \mathbb{F}_{s^{12}}$ and has order s.
- BabyJubjub curve: $E^{\mathsf{BJ}}/\mathbb{F}_s$ defined by equation $E^{\mathsf{BJ}} : 168700x^2 + y^2 = 1 + 168696x^2y^2$, with subgroup $\mathbb{G}^{\mathsf{BJ}}$ of prime order t.
- BW6-761 curve: $E^{\mathsf{BW}}/\mathbb{F}_u$ defined by equation $E^{\mathsf{BW}} : y^2 = x^3 - 1$, with subgroups $\mathbb{G}_1^{\mathsf{BW}}, \mathbb{G}_2^{\mathsf{BW}}$ of prime order v, and an efficiently computable pairing $e : \mathbb{G}_1^{\mathsf{BW}} \times \mathbb{G}_2^{\mathsf{BW}} \to \mathbb{G}_T^{\mathsf{BW}}$, where $\mathbb{G}_T^{\mathsf{BW}} \subset \mathbb{F}_{v^6}$ and has order v as well.
- BLS12-377 curve: $E^{\mathsf{BLS}}/\mathbb{F}_v$ defined by equation $E^{\mathsf{BLS}} : y^2 = x^3 + 1$, with subgroups $\mathbb{G}_1^{\mathsf{BLS}}, \mathbb{G}_2^{\mathsf{BLS}}$ of prime order w, and an efficiently computable pairing $e : \mathbb{G}_1^{\mathsf{BLS}} \times \mathbb{G}_2^{\mathsf{BLS}} \to \mathbb{G}_T^{\mathsf{BLS}}$, where $\mathbb{G}_T^{\mathsf{BLS}} \subset \mathbb{F}_{w^{12}}$ and has order w.

Finite fields.

- $\mathbb{F}_p$: base field of the SECP256K1 curve.
- $\mathbb{F}_q$: scalar field of $\mathbb{G}^{\mathsf{SEC}}$.
- $\mathbb{F}_r$: base field of the BN254 curve.
- $\mathbb{F}_s$: scalar field of $\mathbb{G}_1^{\mathsf{BN}}, \mathbb{G}_2^{\mathsf{BN}}, \mathbb{G}_T^{\mathsf{BN}}$ and base field of the BabyJubjub curve.
- $\mathbb{F}_t$: scalar field of $\mathbb{G}^{\mathsf{BJ}}$.
- $\mathbb{F}_u$: base field of the BW6-761 curve.
- $\mathbb{F}_v$: scalar field of $\mathbb{G}_1^{\mathsf{BW}}, \mathbb{G}_2^{\mathsf{BW}}, \mathbb{G}_T^{\mathsf{BW}}$ and base field of the BLS12-377 curve.
- $\mathbb{F}_w$: scalar field of $\mathbb{G}_1^{\mathsf{BLS}}, \mathbb{G}_2^{\mathsf{BLS}}, \mathbb{G}_T^{\mathsf{BLS}}$.

Generators.

- Generator of $\mathbb{G}^{\mathsf{SEC}}$: XXX.
- Generator of $\mathbb{G}_1^{\mathsf{BN}}$: XXX.
- Generator of $\mathbb{G}^{\mathsf{BJ}}$: XXX.
- Generator of $\mathbb{G}_1^{\mathsf{BW}}$: XXX.
- Generator of $\mathbb{G}_1^{\mathsf{BLS}}$: XXX.

4.2 Hash functions

Marta: Poseidon? Specify parameters? And p may actually not be p, and probably it will be instantiated with different curves, depending on the circuit.

We use three hash functions.

Keccak256. On the one side, the Keccak256 [3] hash function $\mathsf{Hash}_1 : \mathbb{F}_p \to \mathbb{F}_p$ is used to derive an Ethereum address from a public key over $\mathbb{G}^{\mathsf{SEC}}$ (as described in [10]).

Poseidon. On the other side, the hash function we use the most is Poseidon [7] hash function $\mathsf{Hash}_2 : \mathbb{F}_p \to \mathbb{F}_p$, where $\mathbb{F}_p$ is the set of tuples of $\mathbb{F}_p$-elements of any length, and $\mathbb{F}_p$ is the field described in Section 4.1. In particular, we make use of the POSEIDON-128 permutation

$$\pi : \mathbb{F}_p^t \to \mathbb{F}_p^t,$$

where $t = r + c$, with the following parameters:

Rate	Capacity	Full rounds	Partial rounds	S-box
$r = 4$	$c = 1$	$R_F = 8$	$R_P = 59$	$S(x) = x^5$

MiMC-7. Finally, we use the MiMC-7 hash function [1] $\mathsf{Hash}_3 : \mathbb{F}_p \to \mathbb{F}_p$, where $\mathbb{F}_p$ is the set of tuples of $\mathbb{F}_p$-elements of any length, and $\mathbb{F}_p$ is the field described in Section 4.1. This hash is used to hash the public inputs of the different circuits and verify the associated proofs more efficiently (see Section 5.3 for further details).

For the sake of clarity, in our notation we admit anything as an input of the above hash functions, although it might not be parsed as a sequence of $\mathbb{F}_p$-elements. Any bitstring can be converted to a tuple of $\mathbb{F}_p$-elements, so this does not change the actual working of the hash computation. In this document, is implied that the proper conversion happens before feeding the input into the actual hash function.

4.3 Encryption schemes

Old text: Threshold homomorphic encryption, specifically the ElGamal scheme, is used to allow the summation of encrypted votes without decrypting them. This enables the system to compute the final vote tally while maintaining the privacy of individual votes. By ensuring that no vote is exposed during the aggregation phase, this scheme preserves voter confidentiality and provides anti-coercion protection, as voters cannot prove their choice to a third party once an encrypted vote is added.

ElGamal [CITE].

- $\mathsf{Enc}_{\mathsf{pk}}(\mathsf{message} \in \{0,1\}^\star)$:
  1. Map $\mathsf{message}$ to $\mathbb{J}$ via $M = mG$. (link G).
  2. Select a random scalar $k \xleftarrow{\$}$.
  3. Compute $C_1 = k \cdot G \in \mathbb{J}$.
  4. Compute $C_2 = k \cdot \mathsf{pk} + M \in \mathbb{J}$.
  5. Output $\mathsf{ciphertext} = (C_1, C_2) \in \mathbb{J}^2$.
- $\mathsf{Dec}_{\mathsf{sk}}(\mathsf{ciphertext} \in \mathbb{J})$:
  1. Parse $\mathsf{ciphertext} = (C_1, C_2) \in \mathbb{J}^2$.
  2. Compute $K = \mathsf{sk} \cdot C_1 \in \mathbb{J}$.
  3. Output $\mathsf{message} = C_2 - K$.

Marta: The group $\mathbb{J}$ is not the right group. This should be changed after writing the elliptic curves section. Actually, I am unsure we should specify where do all elements live.

Marta: It remains to explain how to recover m from M. I propose to define a function P_m that maps m to $\mathbb{J}$, and then say that although it is generally not invertible, in this case the message space is small, and therefore, it can be done via brute-force search or using the baby-step giant-step algorithm [CITE].

Note that this scheme is additively homomorphic. That is, given two ciphertexts (C_1, C_2) and (C_1', C_2'), their component-wise addition yields: $(C_1^{sum}, C_2^{sum}) = (C_1 + C_1', C_2, C_2)$. The aggregated ciphertext decrypts to the sum of the messages $M^{sum} = M_1 + M_2$.

Marta: The pk used to encrypt is generated by multiple parties as defined in Section 4.5.

Marta: Add reencryption algorithm, which essentially adds $\mathsf{Enc}(0)$: $\mathsf{Enc}(v) = \mathsf{Enc}(v + 0) = \mathsf{Enc}(v) + \mathsf{Enc}(0)$.

4.4 Digital signature schemes

DAVINCI uses the elliptic curve digital signature algorithm (ECDSA) [9] over the SECP256K1 curve to ensure compatibility with standard Ethereum wallets. Verification of ECDSA signatures is *performed inside* zero-knowledge proofs using a specialized circuit that emulates SECP256K1 arithmetic. This approach is necessary because SECP256K1 is defined over a 256-bit prime field that differs from the native field used in the ZK-SNARK circuit (see Section5.3 for more details). Below, we describe the algorithms, which follow the standard ECDSA protocol.

- $\mathsf{S.Sign}_{\mathsf{sk}}(\mathsf{message} \in \{0,1\}^\star)$:
  1. Compute $h = \mathsf{Hash}_1(\mathsf{message}) \bmod q$.
  2. Select a random scalar $k \xleftarrow{\$} \mathbb{Z}_q^*$.
  3. Compute $R = k \cdot G = (x_R, y_R) \in \mathbb{G}^{\mathsf{SEC}}$.
  4. Set $r = x_R \bmod q$. If $r = 0$, go back to step 2.
  5. Compute $s = k^{-1} \cdot (h + r \cdot \mathsf{sk}) \bmod q$. If $s = 0$, go back to step 2.
  6. Output $\sigma = (r, s)$.

- $\mathsf{S.Verify}(\mathsf{pk} \in \mathbb{G}^{\mathsf{SEC}}, \mathsf{message} \in \{0,1\}^\star, \sigma = (r,s) \in (\mathbb{Z}_q^*)^2)$:
  1. Compute $h = \mathsf{Hash}_1(\mathsf{message}) \bmod q$.
  2. Compute $w = s^{-1} \bmod q$.
  3. Compute $u_1 = h \cdot w \bmod q$ and $u_2 = r \cdot w \bmod q$.
  4. Compute
  $$X = u_1 \cdot G + u_2 \cdot \mathsf{pk} = (x_X, y_X) \in \mathbb{G}^{\mathsf{SEC}}.$$
  5. Accept if $r = x_X \bmod q$; otherwise, reject.

4.5 Key generation schemes

Old text: The DKG protocol is used to generate the encryption public key (EPK) in a decentralized manner. Sequencers collaboratively participate in the DKG process to create the EPK, ensuring that no single party has full control over the key. This approach guarantees that the encryption key remains secure and that decryption of results is only possible when a threshold number of sequencers publish their shares.

We use the distributed key generation (DKG) protocol that allows a group of participants to jointly generate a pair of public/private keys for the ElGamal cryptosystem [CITE]. Each participant holds a share of the private key, and only a threshold number of participants can collaborate to decrypt messages.

Marta: We use a variation of DKG because of the encryption of shares. Below I described the original functions but they should be changed!

Marta: Mention that in the SC participants are listed, that is, they have an associated number i.

Let t be the threshold parameter (minimum number of participants required to decrypt messages) and n the total number of participants participating in the DKG protocol ($t \leq n$). Let G be generator point of order q. (This will already be defined in the EC section).

Each participant P_i will use $\mathsf{GenerateShares}$ to generate their share. Then, they will make the set $\{C_i\}_{i=1}^{t-1}$ publicly available and send each s_j privately to each participant s_j. Every participant P_i can verify the share $s_{j,i}$ received from P_j (from of the others) using the algorithm $\mathsf{VerifyShare}$.

- GenerateShares(participant i):
  1. Select random scalars $a_{i,0} \xleftarrow{\$} \mathbb{Z}_q$.
  2. Define the polynomial $f_i(x) = \sum_{j=0}^{t-1} a_{i,j} x^j$.
  3. For every $j \in \{0, \ldots, t-1\}$, compute
  $$C_{i,j} = a_{i,j} \cdot G.$$
  4. For every $j \in \{1, \ldots, n\}$, compute shares
  $$s_{i,j} = f_i(j).$$
  5. Output $\{C_{i,j}\}_{i=0}^{t-1}$ and $\{s_{i,j}\}_{j=1}^{n}$.

Marta: This is not true - according to the specs, shares are encrypted using a simplified version of the EC integrated encryption scheme (ECIES). Each participant generates a zkSNARK proof to prove the correctness of the encrypted shares and compliance of the DKG protocol. This allows the Ethereum SC to verify the validity without revealing the secrets.

Marta: Change the name of algorithm to `GenerateEncryptedShares`.

- VerifyShare($i, j, s_{i,j}, C_i$):
  1. Compute $V = \sum_{i=0}^{t-1} C_i$.
  2. Verify that $V = s_{i,j} \cdot G$.

Marta: This is done by the smart contract - the shares are not in the clear but encrypted.

Marta: Describe slashing mechanism in SC section if one of the shares fails the verification?

Participant P_j uses the algorithm DeriveSecretShare to compute their portion of the collective private key. The algorithm DerivePublicKey generates the public key from the committed data from the participants.

- DeriveSecretShare(participant j):
  1. Compute $s_j = \sum_{i=1}^{n} s_{i,j} \mod q$.
  2. Output s_j.

- DerivePublicKey($\{C_{i,0}\}_{i=1}^{n}$):
  1. Let $pk = \sum_{i=1}^{n} C_{i,0}$.
  2. Output pk.

Note that since $C_{i,0} = a_{i,0} \cdot G$, the public key is effectively sG where $s = \sum_{i=1}^{n} a_{i,0} \mod q$ is the collective private key (unknown to any single participant).

Marta: As before, this should be changed to the encrypted version. It should be explained how the public key is generated from the encrypted shares.

4.6 Merkle trees

Marta: I am not sure a Merkle trees section is needed, but I thought we could use it to specify the hash function used in the trees, the depth, and the -arity. We could also include some functions like `MT.Root()`, etc.

Marta: Sparse MTs (binary) with depth 64. Instantiated with Poseidon hash function.

Marta: Structure of spare Merkle trees from iden3.

4.7 Zero-knowledge proof systems

Old text: zero-knowledge succinct non-interactive arguments of knowledge are a crucial component in ensuring the validity of the voting results. Voters generate zkSNARK proofs to prove that their encrypted votes comply with the rules and requirements of the voting process, without revealing any information about their choices. Sequencers also generate zkSNARK proofs to prove the correct aggregation of votes into the shared state that maintains the status of the voting process.

5 Voting protocol

Marta: Is the process called voting? Election, poll?

5.1 Parties involved

A voting process involves three types of participants: the organizer, the sequencer, and the voters.

Organizer. The organizer is the entity responsible for defining and setting up the voting process. Its key responsibilities include defining the voting parameters and gathering the census data. The organizer ensures that the election is structured correctly but does not participate in vote collection or tallying.

Sequencer. Sequencers are a set of decentralized parties that facilitate the voting process. On the one hand, they collaboratively generate a public encryption key that voters use to encrypt their votes. Then, during the voting phase, they receive and verify encrypted votes from voters and reencrypt votes to prevent coercion.

Voter. Voters are the participants that belong the census that cast their votes in the election.

5.2 Components

5.2.1 Merkle trees *Old text: Merkle trees are employed to create a cryptographic representation of both the voter registry (census) and the voting process state:*

- ***Census**: Each voter is assigned a Merkle proof, which they use to prove their eligibility to vote. This mechanism ensures efficient and secure verification of voter eligibility.*
- ***State**: The voting process state is represented as a Merkle tree, with new votes being added to the tree. The root of this Merkle tree is stored on Ethereum, allowing multiple sequencers to participate in the voting phase and ensuring consistency.*

Census Merkle tree. Explain what it is.

Marta: It could actually be another type of data structure. We only need to be able to get a commitment and a membership proof that can be proven in-circuit. Mention it in some way.

Marta: One way to think about it: (weight, pk).

State Merkle tree. Explain what it is and the structure. Below, the original text:

Marta: Essentially, the first "addresses" or "leaves" are reserved to those items (like processId, censusRoot, etc.), and then the voters' addresses and nullifiers are used as paths to the leave where the encrypted ballot/-commitment is stored.

The State tree contains some special addresses (indices) for storing some required data regarding the voting process:

- Address `0x0`: it stores `processID`, a unique identifier for the voting process.
- Address `0x1`: it stores `censusRoot` and `type`, which contains the necessary information to validate vote proofs.

  Marta: they type was a field in case the organization was using a different structure for the census than a MT.
- Address `0x2`: it stores `ballotMode`, which encodes rules for validating votes, such as the maximum number of selectable options.
- Address `0x3`: it stores the `encryptionKey`, which is the public key used to encrypt the votes.

  Marta: Ideally, EPK is computed before the organization sets the tree, but I am not sure how practical this is.

- Address `0x4`: it stores the `addedResultsAccumulator`. That is, the aggregated encrypted voting results that need to be added.
- Address `0x5`: it stores the `substractResultsAccumulator`. That is, it stores the aggregated encrypted voting results that need to be subtracted.
- Any address: vote `addresses`: are stored within the State tree pointing to a voter's identity commitment. Each commitment is a 32-byte hash derived from the voter's unique information and a secret.

  Marta: (address, commitment).
- Any address: vote `nullifiers`: are stored within the State tree to prevent double voting and to allow vote overwrite. Each nullifier is a 32-byte hash derived from the voter's commitment and a secret.

  Marta: Nullifiers and addresses are used as indexes (i.e. path to the leaf that stores encrypted ballots/commitments).

Marta: Maybe the voteID will be included in the tree at some point.

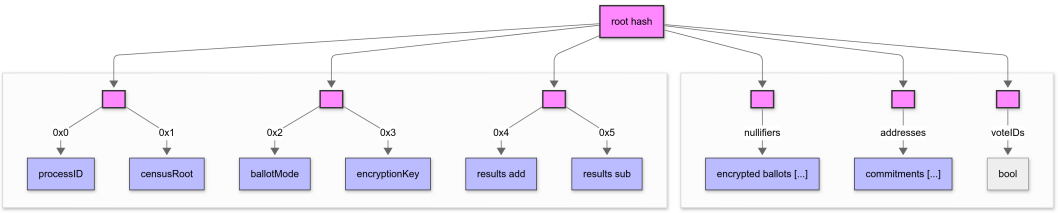


Fig. 3. Structure of the state Merkle tree (FIGURE SHOULD BE POLISHED).

The Initial State: The voting process begins with an initial state where the Merkle Tree Root is established and published on the Process Management smart contract. Predefined parameters are included, but the results are initialized to zero, and no nullifiers are present. The Process Organizer transaction, contains the initial root and the necessary Merkle proofs. These proofs verify that the initial parameters are correct according to the voting process information and that no additional information is stored. Since the 'ProcessId' of the initial state is a unique identifier, there won't be duplicate roots for different processes.

5.2.2 Smart contracts

Process management. This smart contract is responsible for the life-cycle management of voting processes. It includes the initiation, monitoring, execution, and closure of voting events.

Sequencer registry. For each voting process, this smart contract maintains the integrity of the cast votes and process life cycle. It verifies that each state transition committed by a sequencer adheres to the predefined rules.

Results verification. This smart contract keeps track of the existing available sequencers, stores the collateral of the sequencers to ensure (that incentivizes) good behaviour, and is used to coordinate the distributed key generation when a new voting process is created.

Marta: This is wrong. Explanation of SR and RV smart contracts are mixed, I don't know what I was thinking when I wrote it. We should also include a sentence about the management of sequencers.

5.2.3 Votes [I don't think this is the right place] In the DAVINCI system, a vote comprises several components that work together to ensure secure, private, and verifiable voting. These components are:

$\texttt{vote} = [\texttt{processID}, \texttt{censusProof}, \texttt{identityCommitment}, \texttt{nullifier}, \texttt{encryptedBallot}, \texttt{ZKproof}, \texttt{signature}]$.

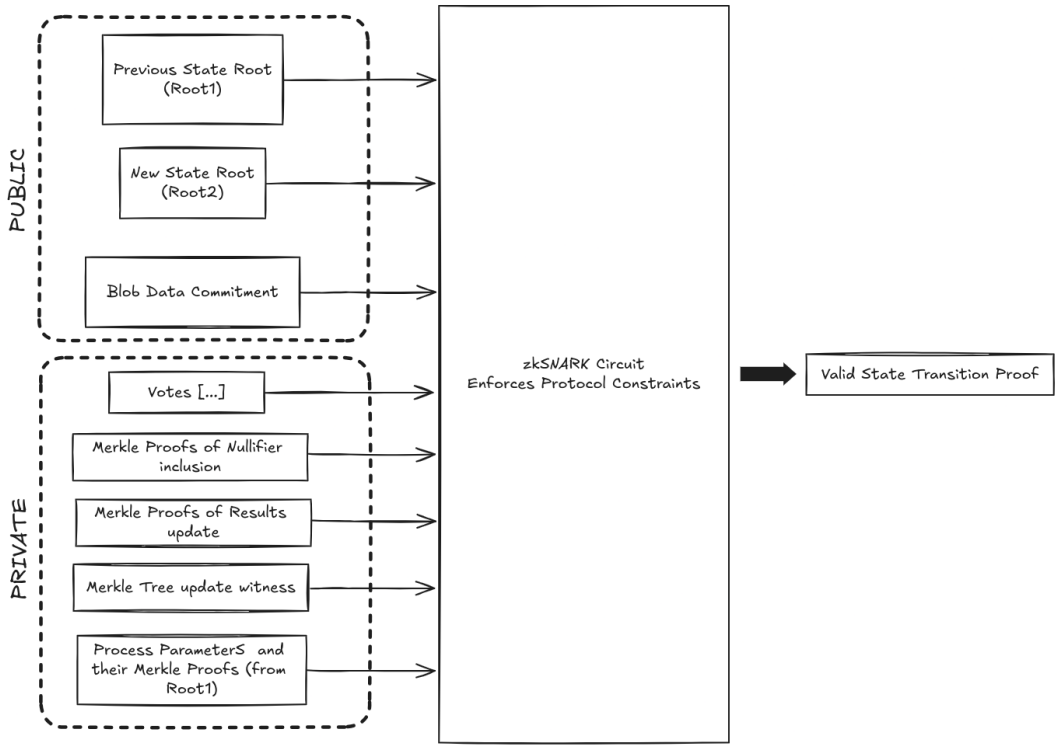


Fig. 4. Caption.

Marta: In the code, the struct is just: address, commitment, nullifier and ballot. Check!

Marta: Actually, the address could be recomputed by the sequencer (add it as a footnote).

Marta: In the code, they call it vote, verifierVote. Alternative: envelope.

Below, we detail each component and its role in the voting process:

- `processID`: the process identifier is a unique 32-byte number that uniquely identifies a specific voting process. This variable encapsulates essential information about the voting, such as the rules and constraints for ballots and the unique identifier of the DAVINCI blockchain instance.

  Marta: the identifier is computed from the organization address, the Eth nonce, chain ID (e.g. mainnet), formatejat in a certain way.

- `censusProof`: the census proof serves as the voter's identity verification mechanism, ensuring that only eligible voters can participate. Depending on the process configuration, the voter provides either a Merkle tree-based proof showing inclusion in the census true, or a credential issued by a trusted third party (e.g. from a credential service provider).
- `identityCommitment`: the identity commitment is used to prevent a voter from registering multiple nullifiers and avoid collisions if different voters choose the same secret. The commitment is computed the following way:

$$\texttt{identityCommitment} = \mathsf{Hash}_2(\texttt{address} \,||\, \texttt{processID} \,||\, s)$$

Marta: Link Hash function to function in Section4.

Old text:

- *Stored in state Merkle tree (SMT): indexed by the voter's address.*
- *Prevents multiple registrations: each address can have only one commitment, ensuring a voter cannot register multiple secrets.*

- *Collision resistance: including the address in* `identityCommitment` *ensures that commitments are unique even if voters choose the same secret s.*

The secret s can be implemented from different ways or a combination of them, such as:

- *A secure enough password introduced by the user.*
- *A signature of a specific text.*
- *A random input that the user must store to allow potential vote overwrite.*

- `nullifier`: the nullifier is a 32-byte hash used to prevent double voting. It is computed the following way:

$$\texttt{nullifier} = \mathsf{Hash}_2(\texttt{identityCommitment}||s)$$

Marta: Link Hash function to function in Section4.

That is, it ensures that each voter can either cast only one vote or overwrite their previous vote. By means of nullifiers, voters can submit a new vote with the same nullifier to replace their previous vote. Moreover, nullifiers provide users' anonymity, since nullifiers cannot be linked back to the voter's identity without knowledge of the secret s.

- `encryptedBallot`: it contains the voter's selections encoded according to the ballot protocol rules. To ensure privacy, the ballot is encrypted using an homomorphic encryption scheme, which allows homomorphic combination of encrypted votes. More specifically, given the voter's choice m, the vote is encoded as follows:

$$\texttt{encryptedBallot} = \mathsf{Enc}(m).$$

- `ZKproof`: the voter generates a ZK proof to prove the correctness of the ballot and encryption process:

$$\texttt{ZKproof} = \text{Prove(), Circuit()}.$$

This proof proves the correct computation of the vote encryption, the nullifier, the commitment, and the compliance with the ballot protocol rules.

- `signature`: the signature authenticates the vote and ensure that was cast by a legitimate voter. The voter signs necessary components using their private key, depending on the census configuration (e.g., ECDSA, EdDSA, RSA).

$$\texttt{signature} = \text{Sign(xxx)}.$$

5.3 Circuits

Marta: The idea of this section is to describe the circuits. Then, in Section 8.2 we give the details such as of the number of constants, framework used, etc. I haven't done this split yet.

Marta: **About the circuits: in practice (and for optimization), all public inputs PI are private, and we produce a single public input $h = Hash(PI)$. Then, the verifier checks that $Hash(PI) = h$ and verifies the proof only against h.** For this, we use the MiMC-7 hash function.

In the case of the voter circuit (circuit 1), the hash h is called the `voteID`, and it allows to voter to trace their vote (it's propagated through the circuits from the sequencer). Since this is an implementation detail but it is used for traceability, I propose to write this as a footnote or as an introduction to this section.

Old text: By structuring the process this way, we ensure that voting can be performed from any device-including smartphones and web browsers-while keeping the sequencer's computational requirements within the capabilities of accessible, CPU-based machines with 64 GiB of memory.

Marta: I don't follow the above paragraph.

Marta: It is called Ballot Mode because we define some modes (like quadratic, etc.) that can be chosen by the organizer. Essentially, each of this modes has a special configuration, but in order to make it easier, the organizer only needs to choose which of the modes (configs) to apply.

Marta: It's convenient to name the proof associated to each circuit: a proof for this circuit is called [...].
Answer: 1. ballotProof, 2. voteVerifier, 3. aggregator, 4. stateTransition.

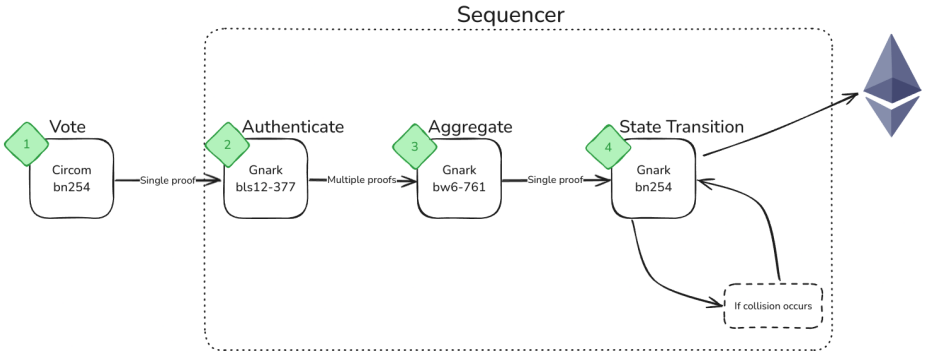


Fig. 5. Circuits flow.

5.3.1 Voter circuit Figure 6 – BN254.

Generated by the user when casting a vote, this circuit proves that the encrypted ballot is valid (i.e. follows the protocol rules) and that the nullifier and commitment are correctly generated.

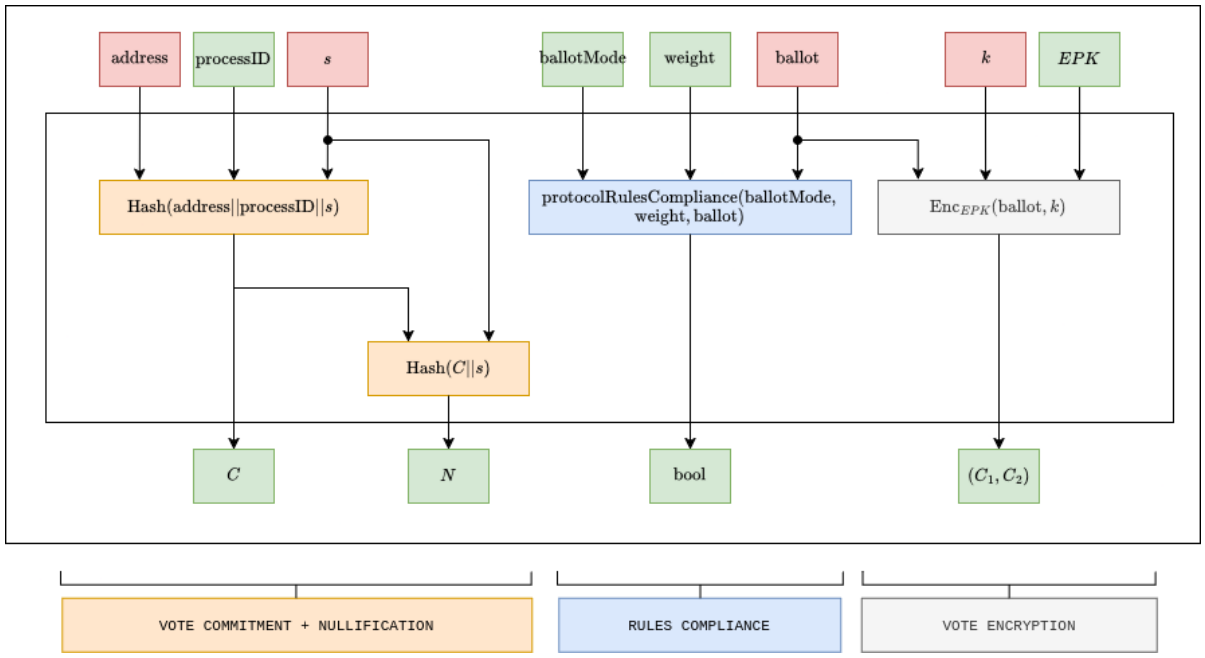


Fig. 6. Voter circuit. All public values are framed in green.

VOTER CIRCUIT

Public inputs

- public `processID`: voting process identifier.
- public `ballotMode`: ballot protocol configuration.
- public `encryptionKey` (EPK): encryption public key.
- public `weight`: voter's weight.
- public C: commitment.
- public N: nullifier.
- public (C_1, C_2): encrypted ballot.

Private inputs

- private `address`: voter's address.
- private `ballot`: plaintext voter's voting choice.
- private s: blinder factor (secret).
- private k: random scalar.

Constraints

Vote encryption.

- Correct encryption of the ballot: $(C_1, C_2) = \mathsf{Enc}_{\texttt{encryptionKey}}(\texttt{ballot}, k)$.

Rules compliance.

- The ballot meets the protocol rules (number of valid choices, etc.): $\ldots + \texttt{weight}$.

Vote commitment + nullification.

- The commitment is correctly computed: $C = \text{Hash}(\texttt{address} \,||\, \texttt{processID} \,||\, s)$.
- The nullifier is correctly computed: $N = \text{Hash}(C \,||\, s)$.
- The secret s links the commitment and the nullifier and the voter has knowledge of such s.

5.3.2 Authentication circuit Figure 7 – BLS12-377.

Generated by the sequencer, this circuit transforms the vote proof to the BLS12-377 curve for native recursion and validates the user's eligibility in the census, as well as their signature.

Marta: Hash of public inputs is in the wrong place. Well, in practice, it will be placed before the verification of the proof, but if we keep the hash of the PI as an implementation details, then it is in the right place. (I propose to keep it like this and in the footnote, mention the changes in each of the circuits).

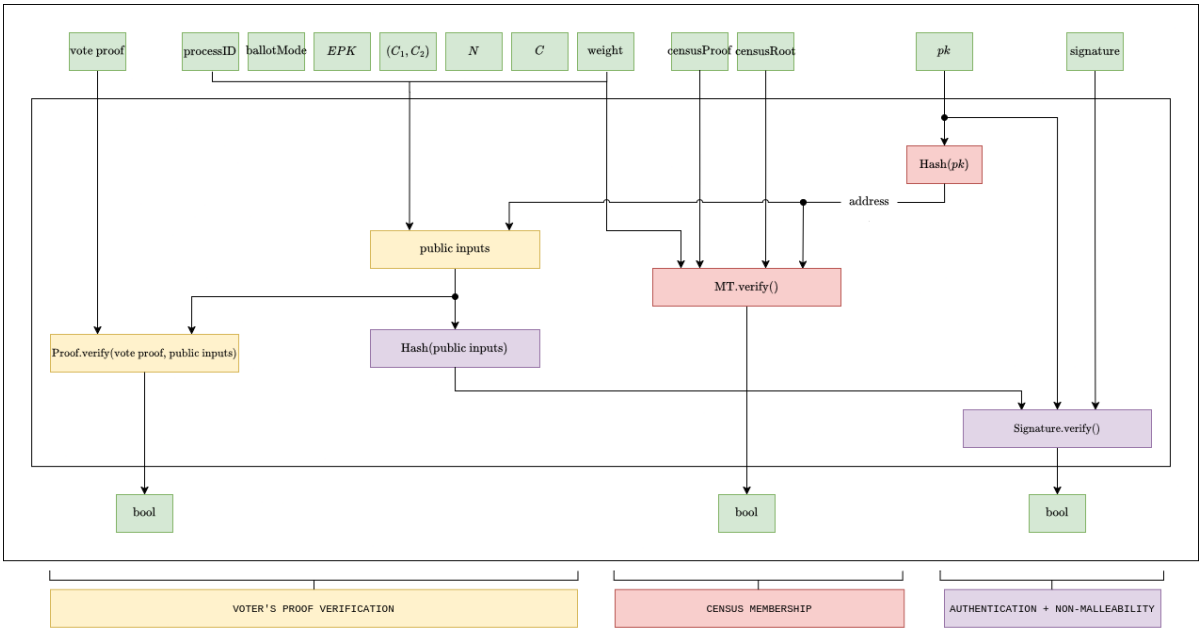


Fig. 7. Authentication circuit. All public values are framed in green.

AUTHENTICATION CIRCUIT

Public inputs

- public `voteProof`: voter's proof.
- public `processID`: process identifier.
- public `ballotMode`: ballot protocol configuration.
- public `encryptionKey` (EPK): encryption public key.
- public (C_1, C_2): encrypted ballot.
- public C: commitment.
- public N: nullifier.
- public `weight`: voter's weight.
- public `censusProof`: census membership proof.
- public `censusRoot`: census Merkle tree root.
- public pk: voter's address.
- public `signature`: voter's signature.

Constraints

Voter's proof verification.

- The vote zkProof is valid for the inputs provided: $P.Verify()$.

Authentication + non-malleability.

- The signature of the inputs provided is valid for the public key of the voter.

Census membership.

- The address derived from the user public key is part of the census, and verifies the census proof with the user weight provided.

5.3.3 Aggregation circuit Figure 8 – BW6-761.

This circuit accumulates multiple authenticated votes into a single proof. It also verifies that all accumulated votes belong to the same voting process.

Marta: This circuit has changed. Now, the aggregation circuit only include the authentication proof and the `voteID`, and then circuit 4 does the "shared public inputs" check and the fact that voteID = Hash(...).

Marta: The batch, number of verification proofs, is a fixed parameter (currently set to 60). If it includes less proofs than the max size of the batch, the sequencer adds dummy proofs to the circuit.

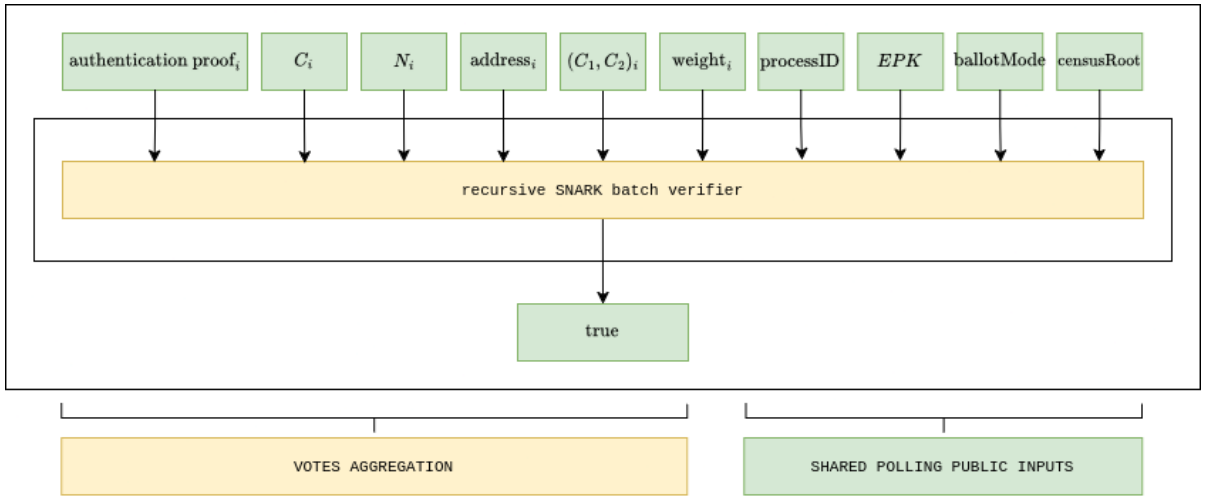


Fig. 8. Aggregation circuit. All public values are framed in green.

AGGREGATION CIRCUIT

Public inputs

- public authenticationProof: set of authentication proofs.
- public C_i: set of commitments.
- public N_i: set of nullifiers.
- public $\texttt{address}_i$: list of voters' addresses.
- public (C_1, C_2): list of voters' encrypted ballots.
- public $\texttt{weight}_i$: Voters' weights.
- public `processID`: process identifier.
- public EPK: encryption public key.
- public `ballotMode`: ballot protocol configuration.
- public censusRoot: census Merkle tree root.

Constraints

Votes aggregation.

- The accumulated zkProofs are valid.

Shared public inputs.

- The ProcessId, CensusRoot, BallotMode and EPK is the same for all of them.

5.3.4 State transition circuit – BN254.

Given the aggregated votes proof, this circuit verifies the correct inclusion of all new votes into the process's state Merkle tree. It generates the final state transition proof that will be validated by the Ethereum smart contract.

Assertions.

- *Aggregation verification*: The agreggated zkProof is valid.
- *Transition verification*: The MerkleTree transition witness proves every change between Root1 and Root2.
- *Public inputs integrity*: ProcessID, BallotMode, CensusRoot, EncryptionKey remain unchanged.
- *Votes counting*: Ballots are correctly counted as new or overwrites, and added to results accumulators.

Marta: The corresponding circuit figure is not done yet.

Old original text, I think it will help here:

When a new voting process begins, the Sequencer initializes a new State, represented by the root hash of a Merkle tree. This tree encapsulates all essential information about the voting process, including process parameters, voter registry (census), ballot configurations, and initial results.

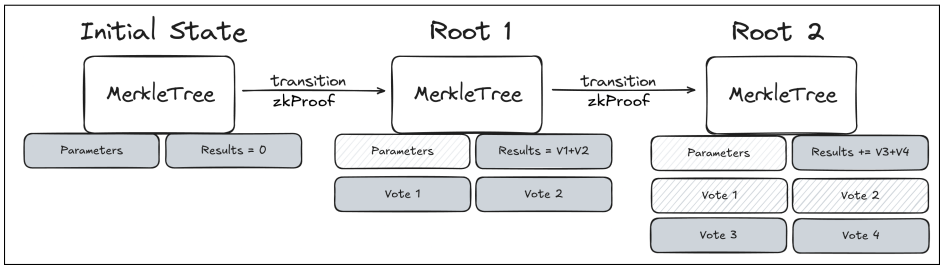
For each new batch of votes, the Sequencer updates the state by generating a zkSNARK proof that **validates the state transition from the current Root to a new Root**. This proof is submitted on-chain for settlement. By doing so, we maintain an immutable and verifiable record of the voting process on the blockchain.

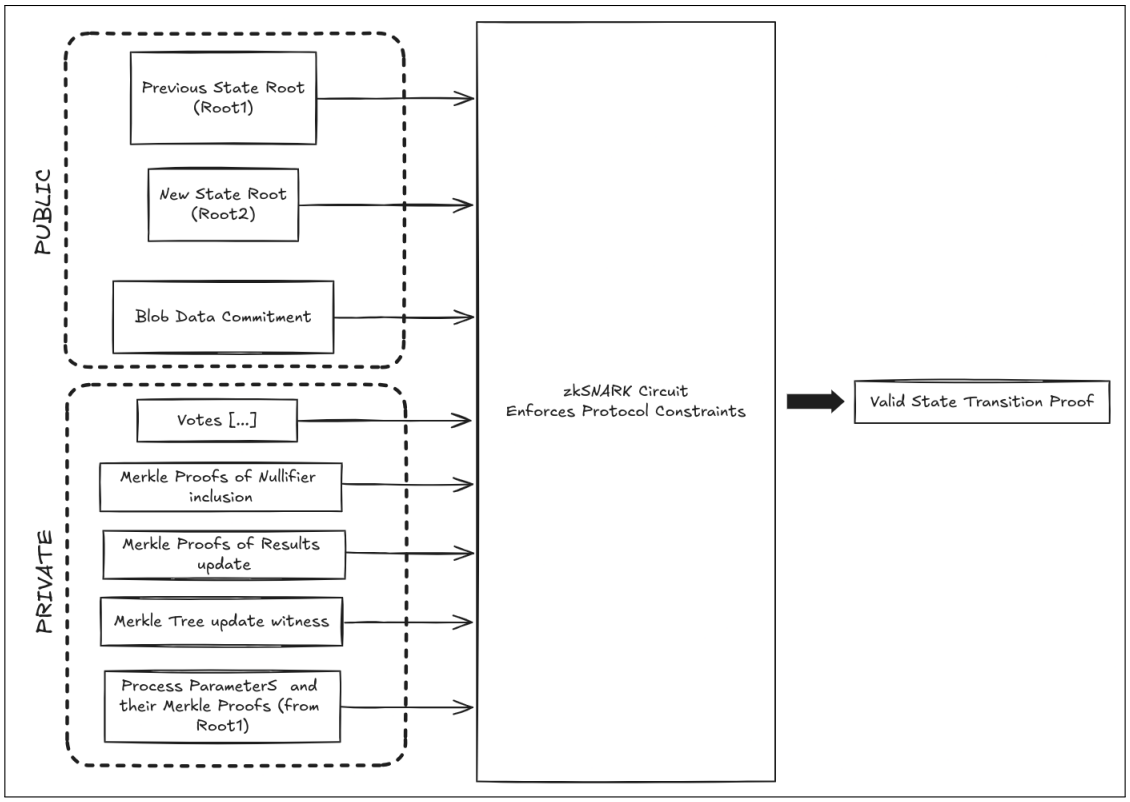
This approach allows anyone to access the latest verified state of the voting process from the Results Verification smart contract, along with the necessary data to process subsequent state updates. The system's design enables multiple Sequencers to participate in tallying votes. They can take the current State Root and its associated data to construct the next state, incorporating new votes into the tally. This decentralization of Sequencers helps prevent potential censorship and reinforces the robustness of the voting process.

Maintaining the Chain. The chain of integrity is maintained through a combination of smart contract enforcement and strict zkSNARK circuit constraints. This ensures that each state transition is valid and builds upon the last accepted state without requiring additional mechanisms.

- **Sequential state roots**: Each state transition updates the Merkle tree from a previous root (‘Root1‘) to a new root (‘Root2‘) after processing a batch of votes.
- **Smart contract enforcement**: The smart contract verifies that the ‘Root1‘ provided in the zkSNARK proof matches the last committed state root stored on-chain. This guarantees that all transitions are sequential and based on the latest accepted state.
- **Proof validation**: The smart contract uses the zkSNARK verification key to validate the submitted proof. A valid proof confirms that the transition from ‘Root1‘ to ‘Root2‘ adheres to all protocol rules enforced by the circuit.
- **State update**: Upon successful verification, the smart contract updates the stored state root hash to‘Root2‘, ensuring an immutable and continuous chain of states.

State Transition To validate and process state transitions, **we employ a zkSNARK circuit that enforces all protocol constraints**. This circuit proves that the transition from the previous state Root to the new state Root is valid based on the newly processed votes.





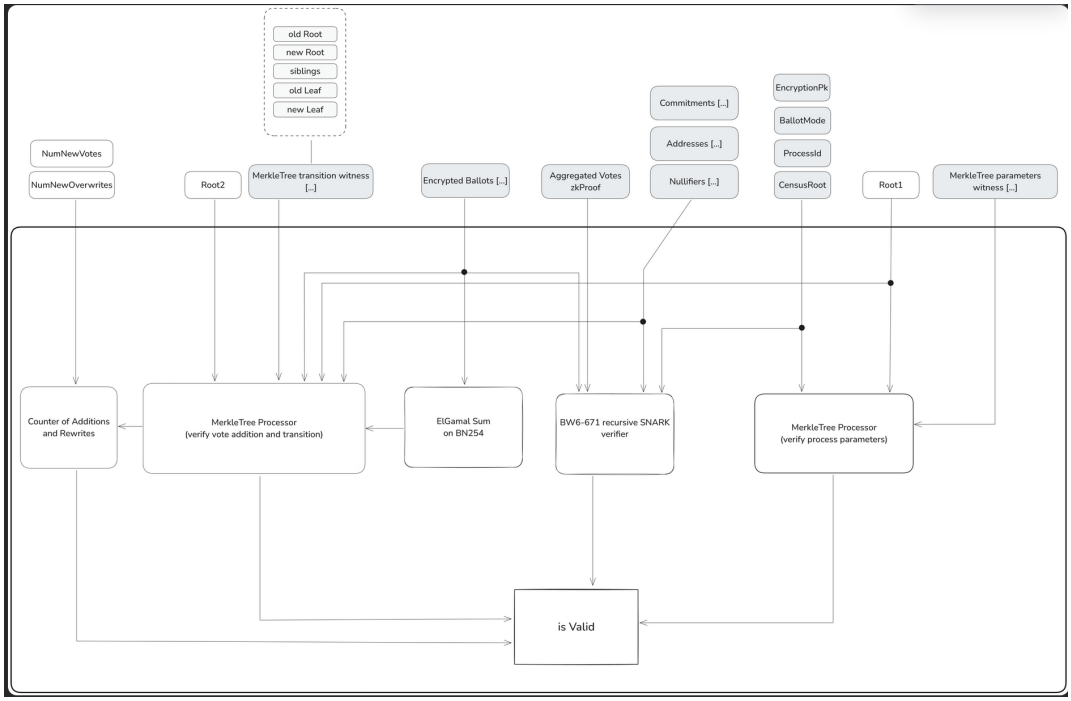


Fig. 9. State transition circuit. All public values are framed in green.

STATE TRANSITION CIRCUIT

Public inputs

- public $root_{old}$: Merkle tree root before the state transition.
- public $root_{new}$: Meklre tree root after the state transition.
- public *blobCommitment*: the commitment to the data blob containing the new votes.

Private inputs

- private Votes: The list of new votes being processed, including census proofs and authentication data.
- private Merkle Proofs of nullifier inclusion: Proofs that each voter nullifier is included in the census.
- private Merkle Proofs of results update: Proofs that the process results have been correctly updated.
- private Merkle Tree Update Witnesses: Necessary data (e.g., Merkle paths) to update the Merkle tree from Root1 to Root2.
- private Process Parameters: Retrieved from Root1 within the circuit.

Constraints

- **Immutable process parameters**: Ensure that critical process parameters (such as 'censusRoot' or 'processId') retrieved from 'Root1' remain consistent and are not altered in the transition.
- **Vote validity**: Validate that each vote proof is correct.
- **Voter eligibility**: Confirm that each voter is included in the census by verifying Merkle proofs of inclusion against the 'censusRoot' retrieved from 'Root1'.
- **Nullifier non-existence**: Ensure that the nullifier for each vote does not exist in the current state ('Root1'), preventing double voting.
- **Nullifier addition**: Correctly add each new nullifier to the state, resulting in 'Root2', updating the Merkle tree accordingly.
- **Results update**: Ensure that the voting results are accurately updated by adding the new votes to the previous results retrieved from 'Root1', so that 'results2 = results1 + votes'.
- **Blob data integrity**: Confirm that the data used in the circuit (votes, nullifiers) corresponds to the 'blobCommitment' provided as a public input, ensuring that the votes processed are exactly those published in the data blob.
- **State transition validity**: Ensure that the new state root ('Root2') is correctly computed from 'Root1' by applying the validated votes and updates to the Merkle tree.

Marta: STATE TRANSITION CIRCUIT: a potential problem is that a sequencer may know if one of their votes has been overwritten, for this reason, sequencers reencrypt a certain amount of ballots every time (already existing ballots I mean). So, the circuit should include the reencryptions: both for existing ballots and also for the new votes, which should be reencrypted (for receipt-freeness).

5.4 Protocol flow

In Figure 10 we describe the protocol flow.

Marta: About the figure: layout should be fixed. In addition, the icons for the different parties are missing.

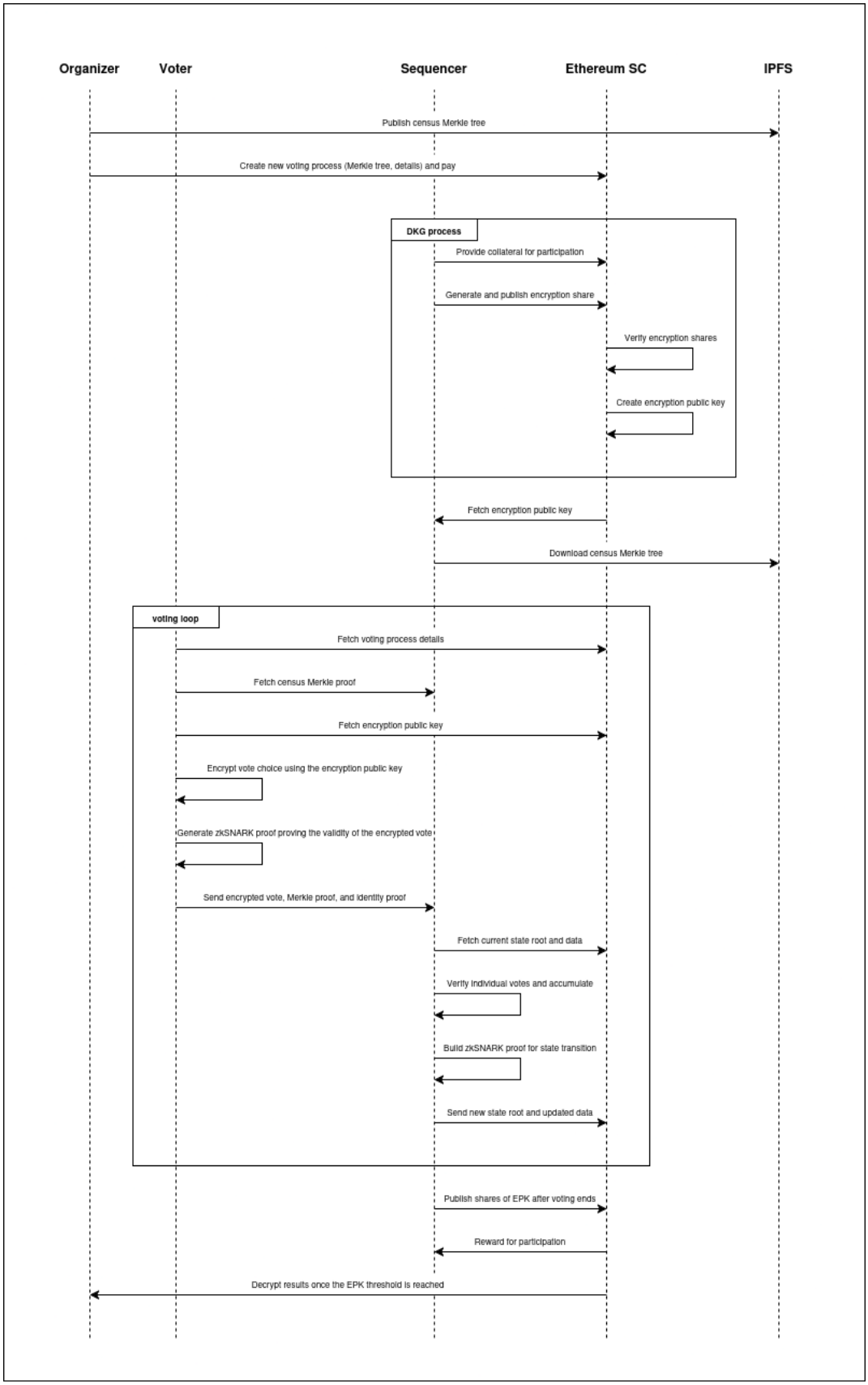


Fig. 10. Vocdoni voting process overview.

5.4.1 Prior to voting (only once)

Marta: Remove the first step, we assume contracts are already deployed or at least, the logic is available.

This is done only once. Then every voting will start its process from step in Section 5.4.2.
Vocdoni:

1. Deployment of the 3 smart contracts.
2. Sequencers registry starts (it doesn't end).

Sequencers (nodes validadors) – this process is always open:

1. Register this and that way.
2. Public keys?

   Marta: Sequencers must also publish how they can be reached (either when they register or when they contribute to the DKG). They can use a URI: Uniform Resource Identifier.
3. PROVIDE COLLATERAL.

   Marta: What happens to the collateral? If done only once, then you lose it? Do you need to send a new tx with more collateral?
4. Send transaction to Sequencer registry smart contract (pay for tx fee).

5.4.2 Start of the voting process

The organizer:

1. Prepares the census data `census_data`.
2. Generates the census commitment `MT_census.root = Com(census_data)`.

   Marta: In general, make use (link) to the functions defined in Section 4. For example, here the `MT.root()`.

   Marta: Additionally, after this step, the organizer should make a Merkle proof available to all eligible voters.
3. Set voting details: duration, options, type, etc. + CENSUS MT ROOT.
4. Also a timeout for the DKG ceremony + minimum number of sequencers.
5. Send transaction to the Ethereum process management smart contract with all data. Note this transaction has a cost (Eth tx fees) that needs to be covered by the organizer.

5.4.3 Key generation process

The sequencer:

1. Download all required data from the PM smart contract which includes the census MT root to handle the new vote.
2. Make contributions to DKG ceremony.

After XXX time (set by the organizer in the PM contract?): all contributions to DKG are done.

Marta: The organizer decides a minimum number of sequencers (security factor).

- EPK "becomes available".

  Marta: I understand EPK becomes available in the SR smart contract, and that key is sent to the PM contract?

  Marta: The idea here is that some sequencer makes EPK available (a.k.a. sends a transaction). Alternatively, generate a SNARK proving the EPK has been generated correctly and the SC verifies it.

5.4.4 Voting process

Voter:

- Choose any of the available sequencers (from?? – from the SR SC).
- Fetch their census **Merkle proof** to prove eligibility (from the census registry).
- Use the EPK to encrypt the voting choice.
- Generate ZK-SNARK of satisfiability of circuit from Fig. 6 with protocol from Section 4.7 to prove the validity of the encrypted vote and adherence to ballot protocol rules. We call this proof **vote proof**.

- Send the encrypted vote, Merkle proof, validity proof, and identity proof to the sequencer.

  Marta: I understand Validity proof refers to “voter circuit"? Merkle proof is for inclusion, right? And identity proof is for example a signature??

  Sequencer:

- Fetch current state: retrieve the current valid process state root from the Etherem smart contract and the associated data from Ethereum blobs.

  Marta: specify which SC.

- Generate a SK-SNARK proof of state transition proving:
  - the validity of the ZK-SNARK proof provided by the voter,
  - the correct accumulation of votes from users, adding them to the process state,
  - the correct sum of the new encrypted votes using the homomorphic properties of ElGamal,
  - the new votes are from eligible users by checking the census Merkle proofs,
  - the new voters have not already voted by checking their nullifiers, or it is a correct vote overwrite,
  - the data blob hash matches the data used to verify the transition.
- Submit updated state: send the new state root to the smart contract and store the updated data in Ethereum blobs.

  Sequencers keep accumulating votes and create state transition until the finalization of the process.

Marta: I understand this finalization is a deadline set by the organizer.

  SC:

- Upon receiving a sequencer’s transaction, the SC verifies the zk-SNARK proof provided byt he sequencer.
- Ensure the origin root corresponds to the current stored state root.
- Confirm that the blob hash matches the one stored in Ethereum.

5.4.5 Results validation Once the voting is completed (I understand the timeout set by the organizer expires), the following things happen.

Sequencer:

- t out of n sequencers publish their shares corresponding to the EPK to the Ethereum smart contract.

  Marta: specify which SC.

- When the threshold of shares is reached, the results can be decrypted by anyone.

  Marta: mention that the SC computes the corresponding ESK, or that can be computed by anyone off-chain, since all info is public.

- Sequencers receive reward for their correct participants depending on the number of sequenced votes.

  Marta: slash if they don’t provide the right share? reward only dependent of sequenced votes? any relation to their participation in the computation of either EPK or ESK?

  This way:

- After the voting period ends and the results are decrypted, anyone can verify the correctness of the final result.

  Marta: explain how.

- Use the zkSNARK state proof and publicly available data on-chain to ensure the integrity and correctness of the entire voting process.

  Marta: elaborate on this.

5.4.6 Finalization of the voting process

6 Ballot protocol

Marta: It is not clear what is the advantage of defining a protocol like this. Is it easy because it is easy to implement as an arithmetic circuit? Make the advantage clear.

Marta: Yes. The advantage is that we can define many types of voting systems (or recounting) with just these few variables. Moreover, we avoid if-else statements, which makes it very convenient to implement in-circuit. More info here: `https://docs.vocdoni.io/architecture/data-schemes/ballot-protocol.html#vocdoni-results-interpretation` (no multiple question supported).

Marta: How is the voter's `weight` encoded as part of the ballot?

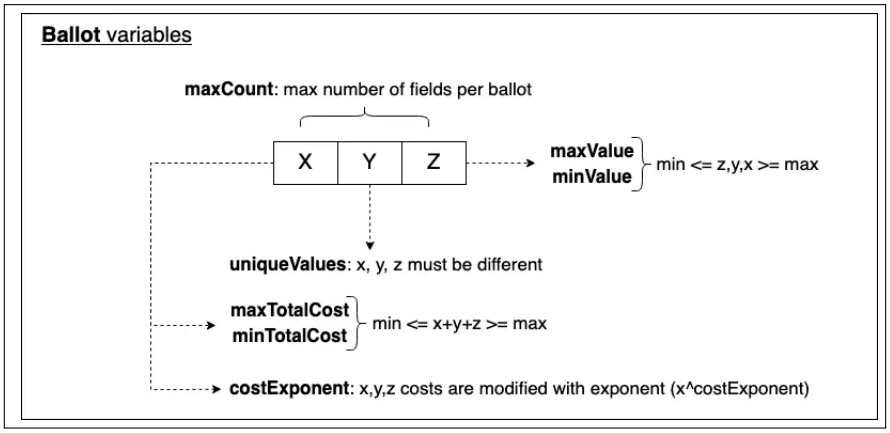
The DAVINCI ballot protocol is a simple and efficient mechanism for casting and tallying votes in any type of election or collective decision-making process. Each voting process consists of one or several fields and voters are required to provide a response for each of these fields in their ballot.

The responses in the ballot are represented as a sequence of natural numbers, each corresponding to the voter's choice for the respective field. Results are accumulated into a single array. Each position in the array corresponds to the sum of all votes cast for that field across all voters.

The ballot protocol is defined by a set of configurable variables that dictate how votes must be cast. This way the protocol can accommodate a wide range of voting processes and behaviors. The list of variables is the following.

1. `maxCount`: it defines the maximum number of fields in a ballot (max 64).
2. `maxValue`: it is the maximum allowable value for any field in a ballot (if greater than 0).
3. `minValue`: it is the minimum allowable value for any field in a ballot (default 0).
4. `uniqueValues`: it specifies whether voters can select the same value multiple times within a ballot (default false).
5. `maxTotalCost`: it limits the sum of all field values in a ballot (if greater than 0).
6. `minTotalCost`: it specifies a minimum required total sum of field values in a ballot (default 0)
7. `costExponent`: it defines the exponent used to calculate the "cost" of votes for each field (default 1).

Marta: "if greater than 0" vs. "default" (make it consistent).



Marta: In Figure: when is says uniqueValues: x, y, z must be different – this is if uniqueValues = false, right?

Example 1: Rating candidates.

Marta: Reorder names Lennon, Hendrix, Joplin alphabetically.

Marta: In this example, the rest of variables are default, right? E.g. minValue = 0?

Consider a voting process where voters are asked to rate three candidates: Lennon, Hendrix, and Joplin. Voters rate each candidate from 0 to 5 stars, and each vote is represented as an array where each position corresponds to the candidate's rating. The configuration of this voting would be the following:

- `maxCount`: 3
- `maxValue`: 5
- `uniqueValues`: Yes

Ballots:

- Vote 1: `[3, 2, 5]` (3 stars for Lennon, 2 stars for Hendrix, 5 stars for Joplin)
- Vote 2: `[4, 3, 2]`
- Vote 3: `[2, 4, 5]`

Marta: I would write 4 ballots.

After accumulating the votes:

- Results Array: `[3+4+2, 2+3+4, 5+2+5] = [9, 9, 12]`
  Lennon received 9 points, Hendrix received 9 points, and Joplin received 12 points.

Marta: In the above example, there is no limit to maxTotalCost, right? Maybe add something like, if maxTotalcost = X, then one of the previous ballots would be invalid (and explain why).

Example 2: Quadratic voting for resource allocation. In a scenario where voters distribute a fixed number of credits across different options (e.g., selecting funding levels for NGOs), the ballot allows voters to assign multiple points, but the cost of casting multiple votes for a single option increases quadratically.

Configuration:

- `maxCount`: 4
- `maxTotalCost`: 12 (credits)
- `costExponent`: 2 (quadratic)

Ballots:

- Vote 1: `[2, 2, 2, 0]`
- Vote 2: `[1, 1, 3, 1]`
- Vote 3: `[0, 2, 1, 2]`

Marta: Add a fourth vote, e.g. [3, 3, 0, 0]. Then mention that the first 3 ballots are ok because $2^2+2^2+2^2=12$, $1^2+1^1+3^2+1^2=12$, $2^2+1+2^2=9<12$, but $3^2+3^2=18>12$, and therefore it's invalid. Did I get this right?

After accumulating the votes:

- Results Array: `[2+1+0, 2+1+2, 2+3+1, 0+1+2] = [3, 5, 6, 3]`
  Each position in the array represents the total sum of credits allocated to each NGO.

Marta: Cost may be dependant on the voter's weight.

7 The Vocdoni token

DAVINCI introduces the Vocdoni token (VOC) as a key element of its decentralized voting ecosystem, playing a crucial role in the protocol's sustainability. The token serves multiple utility functions that align the incentives of all participants (voting organizers, sequencers, and voters) ensuring the integrity, efficiency, and security of the voting system.

7.1 Roles of the VOC

1. **Collateral for sequencers**: Sequencers are required to stake VOC tokens as a collateral to participate in the protocol. This serves as a safeguard to ensure responsible participation. If a sequencer behaves improperly (whether due to malicious intent or unintentional errors) it can face penalties, including the loss of part of its staked tokens.
2. **Incentive mechanism**: Sequencers earn rewards in VOC tokens based on their contribution to processing valid votes and maintaining the network. Rewards are proportional to the number of valid votes successfully added to the shared state.
3. **Payment for voting processes**: Voting processes organizers use VOC tokens to cover the costs of creating and managing voting processes. The costs depend on factors like the size of the voting registry, the voting period's duration, and the desired level of security (based on the number of participating sequencers).
4. **Governance**: The VOC token facilitates decentralized governance by giving the token holders the right to participate in the project governance. Token holders can influence on important matters such as protocol upgrades, ecosystem development, and other initiatives aimed to enhance various aspects of DAVINCI. This ensures that the project evolves in a transparent, community-driven manner.

7.2 Economics for organizers

Organizers of voting processes pay fees in VOC tokens to create and manage their voting events. These fees cover operational costs and incentivize the sequencers.

The costs of voting processes vary based on the following factors:

- **Maximum number of votes**: Larger voter registries require more resources for processing.
- **Voting duration**: Longer voting periods demand extended resource commitments from sequencers.
- **Security level**: Organizers can adjust the number of participating sequencers to balance between cost and security needs.

Fees must be paid upfront but can be partially reimbursed. The formula for calculating the reimbursement is:

$$\mathsf{reimbursement} = \mathsf{totalCost} - \mathsf{totalReward} - \mathsf{baseCost}$$

Given that there is a list of eligible voters for each voting process, organizers should reserve space equal to the maximum number of voters, anticipating that all eligible voters may participate. However, since this is unlikely, a portion of the reward pool may be reimbursed.

The components of this formula are defined in the following sections.

7.3 Voting process cost model

The process cost model defines how the cost for a given process is calculated. Considering sequencer-specific capacities, process duration, number of voters and security costs, we define following components formula:

$$\mathsf{totalCost} = \mathsf{baseCost} + \mathsf{capacityCost} + \mathsf{durationCost} + \mathsf{securityCost}$$

7.3.1 Components definition

- baseCost: The base cost is a fixed fee charged by the sequencer to set up a voting process based on a fixed value plus the number of and a fixed factor. It is independent of the process duration, or the security level. This part cannot be reimbursed thus is a portion of the that will alwways be rewarded to the sequencers.

  $$\mathsf{baseCost} = \mathsf{fixedCost} + \mathsf{maxVotes} \cdot p$$

  Where:
  - fixedCost is a fixed base fee defined into the protocol.

  - maxVotes is the maximum number of votes of a given voting process.
  - p is a small lineal factor defined into the protocol.
- capacityCost: It defines the cost given the current occupancy of the sequencer network. This component models the cost of reserving space for voting processes, relative to the number of sequencers available, the number of running voting processes and the maximum numbers of voters. The cost increases non-linearly as the number available sequencers approaches to the total number of sequencers and the number of running voting processes grows, so when there are fewer sequencers available and big number of voting processes running, the capacity becomes more valuable.

$$k_1 \cdot \left(\frac{\text{totalVotingProcesses}}{\text{totalSequencers} - \text{usedSequencers} + \epsilon} \cdot \text{maxVotes} \right)^a$$

  - k_1: A scaling factor controlling the impact of sequencers usage.
  - totalVotingProcesses: The total number of running voting processes.
  - totalSequencers: The total number of registered sequencers.
  - usedSequencers: The number of sequencers that are already handling other voting processes.
  - a: An exponent that controls how sharply the cost increases as the used capacity approaches the total available capacity. In other words, it controls the non-linearity of the cost increase.
  - ϵ: Is a very small number that avoids zero division.
- durationCost: This component models the cost of running the process for a specific duration, where longer processes incur into more cost. The scaling is non-linear, meaning that shorter processes are more cost-efficient, while longer processes are increasingly expensive.
  It is expected to have a minimum and maximum duration thresholds (e.g 1 hour to 1 year).

$$k_2 \cdot \text{processDuration}^b$$

  - k_2: A scaling factor for the process duration.
  - processDuration: The duration of the process in hours.
  - b: An exponent that controls the non-linear scaling of the process duration.
- securityCost: The security cost models the use of multiple sequencers to ensure a secure process. The cost scales exponentially based on the number of sequencers used, but with diminishing returns as the number of sequencers approaches the total available sequencers:

$$k_3 \cdot e^{c \cdot \left(\frac{\text{numSequencers}}{\text{totalSequencers}}\right)^d}$$

  - k_3: A scaling factor controlling the overall weight of security cost.
  - c: Controls the steepness of the exponential scaling for security costs.
  - numSequencers: The number of sequencers used in this process.
  - totalSequencers: The total number of available sequencers in the network.
  - d: An exponent controlling how fast the security cost increases as the number of sequencers approaches the total available sequencers.

7.3.2 Constraints There is the need to enforce some constrains in the formula in order to avoid impracticable scenarios.

- If processDuration $>$ maxDuration then: totalCost $= \infty$.
- If numSequencers $>$ totalSequencers then: totalCost $= \infty$.

7.4 Economics for sequencers

To become a sequencer and earn rewards, participants must stake VOC tokens as collateral. This collateral is locked during the sequencer registry in the Sequencer Registry smart contract and can be withdrawn upon the sequencer commitments are fulfilled.

Sequencers **receive rewards** based on:

- The number of **votes** they include in the shared state.

- The number of **vote rewrites** they include in the shared state.
  - A vote rewrite can be either a vote overwrite (a voter casting another vote that overwrites the previous one) or a vote re-encryption (made by the sequencer). Rewriting votes enables more flexibility for the voter and is a key mechanism for the receipt-freeness.
  - It is expected for the sequencers to maximize the number of vote rewrites because there is no way to distinguish between a vote overwrite and a vote re-encryption. This is considered a positive behavior because it contributes to the receipt-freeness mechanism of the protocol.
  - A vote and a vote rewrite are distinguishable because the former include a nullifier that has not ever been included in a specific process.
  - Given that each sequencer will submit a ZK proof validating the state transition into the settlement layer smart contract it is straight forward to count the number of votes and the number of vote rewrites processed by each sequencer.
  - Sequencers can allow vote rewrites up to T times the number of new votes for any state transition. T will be defined as a constant in the protocol.
- The number of **non-processed votes** in relation to the maximum number of voters.

The total reward obtained can be expressed as:

$$\mathsf{sequencerReward}_i = R \cdot \left(\frac{\mathsf{votes}_i}{\mathsf{maxVotes}}\right) + W \cdot \left(\frac{\mathsf{voteRewrites}_i}{\mathsf{totalRewrites}}\right)$$

And subject to the constraints:

$$\frac{\mathsf{voteRewrites}_i}{\mathsf{votes}_i} \leq T$$

$$\mathsf{totalReward} = R + W$$

$$R > W$$

- $R > W$ since the sequencers must prioritize processing votes and not be incetivized to just rewrite existing votes.

Where:

- R: A part of the reward pool allocated for a specific voting process.
- votes_i: The number of votes processed by the sequencer for a specific voting process.
- $\mathsf{maxVotes}$: The maximum number of voters participating in a voting process.
- W: A part of the reward pool allocated for a specific voting process.
- $\mathsf{voteRewrites}_i$: The number of vote rewrites processed by the sequencer for a specific voting process.
- $\mathsf{totalRewrites}$: The total number of vote rewrites for a specific process.

7.4.1 Penalties

- Non-participation: Sequencers who fail to meet their obligation, not providing key shares during the tally phase, can have their collateral slashed. Penalites for a sequencer can be expressed as:

$$\mathsf{slashedAmount}_i = s \cdot \mathsf{stakedCollateral}_i$$

Where:
- $\mathsf{slashedAmount}_i$: The total slashed amount for a given sequencer.
- s: The slashing coefficient $0 \leq s \leq 1$.
- $\mathsf{stakedCollateral}_i$: The amount of VOC tokens staked by a given sequencer.

7.5 Summary

The total cost formula combines four main components:

1. A base cost for setting up the process.
2. A capacity cost based on the number of voters, the total running processes and available sequencer capacity.
3. A duration cost based on how long the process lasts.
4. A security cost based on the number of sequencers used, with diminishing returns for using more than a certain number.

The proposed formula ensures that:

- Small processes are cost-efficient.
- Larger processes or processes using a high proportion of available sequencer capacity incur higher costs.
- The cost of security increases rapidly if more sequencers are used, but adding sequencers beyond a certain point leads to diminishing returns.
- Impractical scenarios cannot be reached.

7.5.1 Notes on optimization In the model presented we can clearly see that there are conflicting objectives from the two main actors:

- The organizer (the "buyer" of services) wants to minimize the cost of running the process.
- The sequencers (the "sellers" of capacity) want to maximize their rewards. A sequencer can decide whether to participate based on expected profits.

These are naturally conflicting objectives that can be modeled as a strategic game. However, modeling this equilibrium for the protocol it is not in the scope on this document and will be presented in a separate piece.

8 Analysis

Marta: THE PROTOCOL (does note reveal the vote but it) REVEALS IF SOMEONE HAS VOTED OR NOT.

Marta: Check EthResearch post: https://ethresear.ch/t/vocdoni-protocol-enabling-decentralized-voting-for-the-masses-with-zk-technology/21036.

Marta: This section is old text, it needs to be reviewed.

8.1 Security discussion

Make a section called *properties*?

8.1.1 Receipt-freeness In democratic processes, the ability of voters to cast their votes freely, without undue influence or coercion, is crucial. A critical aspect of maintaining this freedom is ensuring receipt-freeness, preventing voters from being able to prove to third parties how they voted. DAVINCI implements receipt-freeness by leveraging the properties of the ElGamal cryptosystem, and zkSNARKs.

Ballor re-encryption. The ElGamal cryptosystem over elliptic curves supports additive homomorphism, allowing operations to be performed on ciphertexts that translate to addition on the underlying plaintexts. Re-encryption is a process that refreshes the randomness of a ciphertext without changing the underlying plaintext message. This makes computationally infeasible to link the original and re-encrypted ciphertexts.

Handling receipts. To prevent voters from being able to prove how they voted, DAVINCI employs re-encryption of ballots by the Sequencers. When a voter submits an encrypted ballot, the Sequencer re-encrypts it before storing it in the state Merkle tree.

This way, the system ensures that voters cannot produce a receipt of their vote by revealing r since the stored ciphertext no longer corresponds to r. This prevents vote-buying and coercion by third parties.

Handling collusion. To further enhance receipt-freeness and mitigate the risk of collusion between voters and Sequencers, DAVINCI allows voters to overwrite their votes. A voter can submit multiple votes, with each new submission replacing the previous one.

When a Sequencer receives a new vote from a voter who has already voted, it performs the following steps:

1. **Detect overwrite**: The Sequencer checks the state Merkle tree using the voter's nullifier to determine if the voter has previously cast a vote.
2. **Subtract previous vote**: The Sequencer takes the existing encrypted ballot and adds it to a "Subtractive Results" accumulator. This effectively negates the previous vote in the final tally.
3. **Add new vote**: The new encrypted ballot is added to the "Results" accumulator.
4. **Update state Merkle tree**: The Sequencer re-encrypts the new ballot and updates the state Merkle tree with this re-encrypted ballot.

Concealing vote overwrites. To prevent observers from detecting when overwrites occur, the Sequencers regularly re-encrypt a random subset of ballots in the state Merkle tree during each state update. This process obscures the occurrence of overwrites, as re-encryptions are indistinguishable from standard re-randomizations performed for privacy enhancement.

By re-encrypting random ballots, the system increases the entropy and makes it statistically improbable for an adversary to determine if a specific ballot was overwritten or simply re-randomized.

8.1.2 Privacy DAVINCI ensures user anonymity by anonymizing both the ballot and the voter's identity, employing cryptographic techniques that maintain privacy even in the face of future quantum computing threats. The system uses the ElGamal homomorphic encryption scheme to encrypt ballots, allowing for the aggregation of votes without revealing individual choices. Since encrypted ballots are stored in public repositories like Ethereum blobs—which, although removed after some weeks, may still be accessible—it is crucial to prevent any association between decrypted ballots and voter identities. By decoupling the voter's identity from their encrypted ballot through the use of secrets and cryptographic hashes, even if an adversary decrypts the ballots in the future, they cannot link them back to individual voters.

The identity anonymization acts as a double security factor, enhancing long-term privacy. While the Sequencer processing the vote can identify the voter (since voters submit proofs of eligibility and commitments), there is no incentive for the Sequencer to make this information public, and the Sequencer cannot decrypt the voter's ballot because they do not possess the private decryption keys. This ensures that voter choices remain confidential.

We have adopted this partial identity anonymization because generating fully anonymous proofs using zkSNARKs directly from digital signatures (e.g., ECDSA/EdDSA or RSA) is computationally intensive for client-side devices like browsers and smartphones. Our priority is to support a wide range of devices, making the system accessible to as many voters as possible. In the future, as cryptographic technology advances and client devices become more powerful, we anticipate being able to generate such zero-knowledge proofs efficiently on the client side. This would enable us to fully anonymize the client's identity in addition to the ballot, further enhancing user privacy without compromising accessibility or user experience.

8.1.3 Quantum resistance As quantum computing technology advances, it poses significant challenges to classical cryptographic schemes that underpin the security of digital systems, including voting platforms like DAVINCI. Ensuring that the system remains secure in the face of quantum threats is crucial for its longevity and trustworthiness.

Quantum computers have the potential to solve certain mathematical problems exponentially faster than classical computers. Notably, Shor's algorithm allows quantum computers to efficiently factor large integers and compute discrete logarithms, undermining the security of widely used cryptographic schemes such as RSA, DSA, ECDSA, and the ElGamal cryptosystem.

However, the design of DAVINCI incorporates mechanisms that preserve voter anonymity even in the face of future quantum attacks:

- Detachment of identity and encrypted Ballot: The voter's identity and their encrypted ballot are decoupled through the use of a secret in the nullifier. The nullifier is computed as:

$$N = \text{Hash}(\texttt{processID} \,||\, s)$$

where is a secret known only to the voter. This means that even if an adversary decrypts the encrypted ballots using a quantum computer, they cannot link a decrypted vote back to a voter's identity in the census without knowledge of the secret s.

Marta: Instead of "detachment", I propose "unlinkability": identity unlinkability.

- Quantum-resistant hash functions: the `nullifier` and `identityCommitment` are computed using cryptographic hash functions that are believed to be resistant to quantum attacks (e.g., SHA-3). While Grover's algorithm can provide a quadratic speedup in searching for preimages, using sufficiently long hash outputs (e.g., 256 bits) mitigates this risk.

To further safeguard against quantum threats, the following measures can be implemented in the future:

1. Adopt post-quantum signature schemes. Replace ECDSA/EdDSA with quantum-resistant algorithms such as CRYSTALS-Dilithium, Falcon, or Rainbow, which are based on hard lattice problems.
2. Explore lattice-based homomorphic encryption schemes. Replace ElGamal cryptosystem with quantum-resistant alternatives such as the Brakerski-Gentry-Vaikuntanathan (BGV) or the Brakerski/Fan-Vercauteren (BFV) schemes.
3. Adopt quantum-resistant zero-knowledge proof systems, such as zkSNARK constructions based on post-quantum assumptions or **zkSTARKs**.

Marta: Citations to protocols are missing.

Marta: If it is not quantum resistant, I would leave it to future work. Although unconventional, we could also make a "protocol weaknesses" section.

8.1.4 Data availability

8.2 Implementation

Marta: Link to repos, etc. + details such as circuits have been implemented using `circom`, this and that.

Circuits.

- Circuit 1: circom/snarkJS, $\sim$ 53.000 constraints.
- Circuit 2: gnark, $\sim$ 3.1 million constraints.
- Circuit 3: gnark, 40.000 $\times$ (number of votes) constraints.
- Circuit 4: gnark, $\sim$ 16 million constraints.

8.3 Performance evaluation

Marta: E.g. circuit constraints benchmarks.

9 Conclusions

10 Future work

- Voter: they do circuit1 + circuit2 themselves (instead of the sequencer doing circuit2).
- Post-quantum.
- Support to xxx.

Acknowledgments

The authors would like to thank the following reviewers and contributors for their valuable feedback and support: all team members from the Vocdoni association, Jordi Baylina (from Iden3 and Polygon), Adrià Maçanet (Privacy Scaling Explorations, Ethereum Foundation), Arnaucube (0xPARC), Alex Kampa (AZKR), Roger Baig (Polytechnic University of Catalonia), Javier Herranz (Polytechnic University of Catalonia), and Jordi Puiggali (Secrets Vault).

References

1. Albrecht, M., Grassi, L., Rechberger, C., Roy, A., Tiessen, T.: Mimc: Efficient encryption and cryptographic hashing with minimal multiplicative complexity. In: Cheon, J.H., Takagi, T. (eds.) Advances in Cryptology – ASIACRYPT 2016. pp. 191–219. Springer Berlin Heidelberg, Berlin, Heidelberg (2016)
2. Bellés-Muñoz, M., Whitehat, B., Baylina, J., Daza, V., Muñoz-Tapia, J.L.: Twisted edwards elliptic curves for zero-knowledge circuits. Mathematics **9**(23) (2021). https://doi.org/10.3390/math9233022, https://www.mdpi.com/2227-7390/9/23/3022
3. Bertoni, G., Daemen, J., Peeters, M., Van Assche, G.: The KECCAK SHA-3 submission (2011), https://keccak.team/files/Keccak-submission-3.pdf
4. Bowe, S., Chiesa, A., Green, M., Miers, I., Mishra, P., Wu, H.: ZEXE: enabling decentralized private computation. In: 2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020. pp. 947–964. IEEE (2020). https://doi.org/10.1109/SP40000.2020.00050, https://doi.org/10.1109/SP40000.2020.00050
5. Brown, D.R.L.: SEC 2: Recommended elliptic curve domain parameters. In: Standards for efficient cryptography 2 (SEC 2) (2010), https://www.secg.org/sec2-v2.pdf
6. El Housni, Y., Guillevic, A.: Optimized and secure pairing-friendly elliptic curves suitable for one layer proof composition. In: Krenn, S., Shulman, H., Vaudenay, S. (eds.) Cryptology and Network Security. pp. 259–279. Springer International Publishing, Cham (2020)
7. Grassi, L., Khovratovich, D., Rechberger, C., Roy, A., Schofnegger, M.: Poseidon: A new hash function for zero-knowledge proof systems. In: 30th USENIX Security Symposium (USENIX Security 21). pp. 519–535 (2021)
8. Jancar, J.: Standard curve database: BN254 (2020), https://neuromancer.sk/std/bn/bn254
9. Johnson, D., Menezes, A., Vanstone, S.: The elliptic curve digital signature algorithm (ECDSA). Int. J. Inf. Secur. **1**(1), 36–63 (Aug 2001). https://doi.org/10.1007/s102070100002, https://doi.org/10.1007/s102070100002
10. Wood, G., et al.: Ethereum: A secure decentralised generalised transaction ledger. Ethereum project yellow paper **151**(2014), 1–32 (2014)