DAVINCI: Decentralized autonomous vote integrity network with cryptographic inference

Pau Escrich[1], Marta Bellés-Muñoz[2], Jordi Piñana[1], Lucas Menéndez[1], Alex Kampa[1], Roger Baig[3], and Jose Luis Muñoz-Tapia[3]

[1] Vocdoni Association
[2] Pompeu Fabra University
[3] Polytechnic University of Catalonia

{pau,painan,lucas,alex}@vocdoni.org, marta.belles@upf.edu
{roger.baig,jose.luis.munoz}@upc.edu

Last update: March 26, 2026

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. Using 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 such as the 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 election processes. The protocol employs distributed key generation among sequencers, coordinated through Ethereum smart contracts, and uses Ethereum data blobs for data availability. With a focus on accessibility, scalability, receipt-freeness, and automation, DAVINCI aims to facilitate high-frequency and low-cost voting, fostering mass adoption of voting and simplifying civil participation. Moreover, the introduction of the Vocdoni token aligns incentives among participants, ensuring the system's sustainability and enabling decentralized governance. Finally, the design of DAVINCI is grounded in practicality; All components are implemented using current technologies and have undergone proof-of-concept testing. In sum, the proposed architecture is not merely theoretical, but a viable solution ready for short-term deployment. The source code has been made open source, allowing practitioners and researchers to further investigate its details and potential for future uses.

Table of Contents

1 Introduction

Online voting remains one of the most studied yet elusive applications in applied cryptography. As digital services expand across both public and private sectors, secure and universally verifiable online voting systems have become an increasingly desirable goal. Remote electronic voting promises scalability, accessibility, and administrative efficiency; yet the design of systems that simultaneously ensure privacy, integrity, coercion resistance, and transparency under realistic adversarial models remains a formidable challenge. Numerous deployments have revealed deep-rooted usability flaws and security gaps, particularly when verifiability mechanisms are either poorly understood by users or reliant on centralized infrastructure.

Against this backdrop, the Vocdoni project was initiated in 2018 with the aim of rethinking online voting from first principles. The name Voĉdoni, meaning *to give voice* in Esperanto, reflects the project's foundational goal: to empower collectives, from small associations to millions of citizens, to engage in secure and verifiable decision-making, regardless of technological or institutional barriers. Central to this vision was the idea that voting is not limited to formal governmental elections but serves as a more general-purpose mechanism for collective signaling. Vocdoni introduced a fully anonymous end-to-end verifiable voting system designed to operate efficiently on a range of devices, including smartphones. To support these goals, the team deployed a custom infrastructure emphasizing resilience, neutrality, and transparency.

Technically, the architecture of Vocdoni was based on a bespoke byzantine fault tolerant (BFT) layer-1 blockchain, named Vochain. At the time, efficient zero-knowledge (ZK) proof systems were emerging, but not yet practical for most deployments. Vochain provided a performant and low-cost environment (achieving approximately 700 transactions per second) in which advanced cryptographic tools could be used without the constraints imposed by general-purpose blockchains based on the Ethereum virtual machine (EVM). The ability to issue voting transactions without requiring user fees enabled broader accessibility. Over several years of development and deployment, this architecture proved both viable and valuable in practice. However, broader adoption as a universal voting protocol highlighted the need for further architectural refinements and stronger formal guarantees.

In this work, we introduce DAVINCI, a new protocol that builds upon the lessons and conceptual groundwork laid by Vocdoni. DAVINCI adopts a modular design and integrates state-of-the-art cryptographic tools, including succinct ZK proofs, improved bulletin board constructions, and robust coercion resistance mechanisms, reflecting the most recent advances in academic research. Unlike monolithic designs, DAVINCI is conceived as a composable primitive: a foundational layer intended to support secure and verifiable voting in diverse contexts, from blockchain governance to institutional elections. This shift in design philosophy aims to address long-standing challenges in the field, offering a cleaner abstraction with clearer security boundaries and formal underpinnings.

1.1 Contributions

This work introduces DAVINCI, a modular and verifiable protocol for digital voting, designed to support privacy-preserving, auditable, and adaptable election processes across diverse governance contexts. The protocol *models voting as a constrained state machine*, in which each valid operation is expressed as a formally defined state transition, enforced through succinct ZK proofs. That is, at the core of the system lies a set of reusable arithmetic circuits that implement voter authentication, encrypted ballot validation, state transitions, and tally finalization. These circuits are optimized for off-chain proving and are compatible with modern zero-knowledge succinct non-interactive argument of knowledge (ZK-SNARK) protocols. Application-layer state is maintained off-chain and committed on-chain using Merkle trees as cryptographic accumulators.

A key contribution of this work is the introduction of a generic ballot protocol that abstracts a wide range of voting systems into a unified, parameterized representation. Instead of designing specialized circuits for each voting rule, ballots are expressed as fixed-length vectors subject to configurable constraints, enabling support for approval, ranking, quadratic, single-choice, and multiple-choice elections within the same circuit framework. This design significantly reduces circuit complexity while allowing different tallying and counting rules to be enforced efficiently inside ZK proofs.

To coordinate protocol execution over a decentralized network of nodes, DAVINCI includes a set of Ethereum smart contracts that mediate the setup of elections, manage finalization procedures, and verify

submitted proofs of correctness. A standardized transaction and data encoding format supports all stages of the process, including vote casting, ballots aggregation, and result publication. This approach ensures that protocol interactions remain deterministic and interoperable across clients and backends. The system also introduces the Vocdoni token, which is used to align incentives between participants. Altogether, the design aims to balance auditability and privacy while reducing trust assumptions on infrastructure providers. By cleanly separating functionalities and providing modular interfaces, the protocol is intended to serve as a foundation for both practical deployments and future extensions, such as integration with alternative identity systems, off-chain data availability layers, or other consensus backends.

1.2 Related work

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

1.3 Paper organization

The rest of the paper is organized as follows. In Section 2 we introduce the necessary background. In Section 3 we provide a high-level overview of a voting process, omitting technical details for clarity. In Section 4 we describe the full voting process in detail and in Section 5 we focus on the ballot protocol. In Section 6 we discuss the role of the Vocdoni token within the incentive structure of the system. In Section 7 we analyze the protocol, covering its security properties, implementation details, and performance evaluation. In Section 8 we conclude the paper with final remarks and future work. Finally, we include Appendix A with the concrete instantiations of all the cryptographic primitives used in the protocol.

2 Background

The DAVINCI protocol builds on several decentralized technologies and advanced cryptographic tools. In this section, we give a high-level overview of these components and their role in the system. We first introduce the interplanetary file system (IPFS), which serves as a decentralized storage layer for distributing large datasets off-chain. We then describe Ethereum, the settlement layer where commitments are published, rules are enforced through smart contracts, and state data is managed using recent mechanisms such as data blobs. Finally, we present the ZK proof systems that underpin the protocol's integrity and security, focusing on ZK-SNARKs and arithmetic circuits, which allow participants to prove compliance with protocol rules without revealing sensitive information.

2.1 Interplanetary file system

The interplanetary file system (IPFS) is a peer-to-peer distributed storage network designed to make the web more resilient, permanent, and decentralized. Unlike traditional client-server architectures that retrieve data from a specific location (e.g., a URL pointing to a server), IPFS retrieves data based on its content. Every file stored in IPFS is split into blocks, each block is hashed, and the resulting cryptographic hash is used as its unique identifier. This property, known as content addressing, ensures that data is tamper-evident: if the file changes, so does its hash. IPFS also provides deduplication (the same content is only stored once across the network), versioning (content identifiers can point to immutable snapshots of data), and distributed availability (data can be fetched from any node that stores it). Together, these features allow IPFS to function as a decentralized content delivery network.

In the context of the DAVINCI voting protocol, IPFS is particularly useful for off-chain data distribution. Large datasets such as the full census (eligible voters and their weights) and election metadata (which may include questions, options, images, etc.) do not need to be stored directly on Ethereum. Instead, they are stored in IPFS, and only their content hashes are published on-chain. This approach ensures transparency and verifiability, as anyone can fetch the dataset from IPFS and recompute the hash to confirm its integrity, while keeping on-chain storage costs low. That is, election participants can therefore rely on IPFS to access the data without burdening the blockchain with large amounts of data.

2.2 Ethereum

Ethereum is a decentralized blockchain platform that supports programmable transactions through its built-in execution environment, the Ethereum virtual machine (EVM). All interactions with the Ethereum network take the form of transactions, which must be broadcast to the network, validated by consensus, and permanently recorded on-chain. Every transaction requires the payment of a fee (gas), denominated in ETH, to compensate validators for computation and storage. This fee model ensures that resources are used efficiently and prevents denial-of-service attacks by making large or complex operations costly. In the context of DAVINCI, Ethereum serves as the settlement layer where critical commitments are published, ensuring transparency and immutability.

Smart contracts. Smart contracts are programs deployed on the Ethereum blockchain that execute deterministically in response to transactions. Once deployed, they cannot be altered, and their execution is guaranteed by the consensus of the network. In DAVINCI, smart contracts orchestrate the election by managing state transitions and storing critical data such as the current state root and the encryption public key. They enforce the protocol rules in a trustless environment, ensuring that no single participant can manipulate the election. By anchoring these rules in Ethereum smart contracts, DAVINCI guarantees that the election logic is applied consistently and verifiably across all participants.

Data blobs. Data availability is a key requirement for decentralized protocols. Ethereum's recent EIP-4844 [9] introduces data blobs as a mechanism for publishing large volumes of off-chain data alongside transactions at a lower cost than traditional storage. These blobs are kept on-chain for 4096 epochs, approximately 18 days, after which they are pruned. For longer election periods, third-party solutions based on EIP-4844 can be used to extend data availability [1]. In DAVINCI, sequencers use data blobs to share state transition data efficiently. By publishing state updates in blobs, sequencers allow other participants to reconstruct and verify the evolution of the election without overloading Ethereum's permanent storage. This ensures scalable, decentralized data availability while retaining verifiability.

2.3 Merkle trees

Merkle trees [18] are cryptographic hash trees that enable compact proofs of data inclusion. Formally, a Merkle tree is a binary tree structure where each leaf node represents the cryptographic hash of a set of data and every non-leaf node is derived from the hash of its children. The apex of the tree, the *Merkle root*, acts as a succinct cryptographic commitment to the entire dataset. To prove that a specific element x exists within the set, a prover provides a Merkle path consisting of the sibling nodes along the path from the leaf to the root. A verifier can reconstruct the root in logarithmic time $O(\log n)$ and check it against the committed state, without requiring access to the full dataset. In DAVINCI, we use two specialized variants of this structure to commit critical data: a sparse Merkle tree for the voting state and an incremental Merkle tree for the voter registry (census).

Sparse Merkle trees. A sparse Merkle tree (SMT) represents a key-value map where each leaf is the hash of a (key, value) pair and each parent node is the hash of its two children. In an SMT, every possible key maps to a unique leaf position (with empty slots defaulting to a zero value), yielding a fixed-height binary tree. This property is circuit-friendly, since the proof of a leaf's membership or non-membership has a consistent length and can be efficiently verified inside a ZK-SNARK circuit using a SNARK-optimized hash. In DAVINCI, the state tree is implemented as an SMT of fixed depth. In particular, configuration parameters occupy reserved leaf addresses, each voter's encrypted ballot is stored at a leaf indexed by their unique identifier, and each one-time vote identifier is recorded at a dedicated leaf path. Because the SMT covers a vast key space, a user or sequencer can prove that a given key is present in the state (or conversely, that it remains empty) by providing a short Merkle proof against the tree's root, rather than revealing the entire state. This enables compact proofs of state correctness. For instance, showing in ZK that a vote identifier has not appeared before or that a new ballot's ciphertext is correctly inserted into the state. The SMT used in DAVINCI follows the circomlib Merkle tree specification (Poseidon-based) [1], ensuring that state updates can be verified succinctly on-chain via the Merkle root and proof.

Incremental Merkle trees. The census (voter registry) is maintained with an incremental Merkle tree (IMT), a binary Merkle tree designed for efficient sequential updates [1]. Unlike a sparse tree, the IMT grows in height only as needed with the number of leaves and eliminates the overhead of hashing dummy "zero" siblings. In this scheme, voters are assigned sequential leaf indices in a continuously evolving tree, rather than being placed at cryptographic key-derived positions. In DAVINCI, the census Merkle tree is built as an IMT: each eligible voter's identifier (or a commitment to it) and voting weight are stored in the next available leaf, and only the Merkle root of this tree (the `censusRoot`) is published on-chain. This design makes membership updates and proof generation extremely low-cost for the census. Appending a new voter requires recomputing hashes only along the single path from the new leaf to the root, and the tree's depth adjusts optimally (e.g. a tree of N voters has height $\approx \log_2 N$, instead of a fixed large height). Consequently, proving one's inclusion in the voter list is more efficient than it would be with an SMT: the Merkle proof is shorter and involves no unnecessary default nodes. A voter can thus obtain a small proof of their membership in the census (their leaf's existence under the known root) without revealing their index or identity, and include this in a ZK-SNARK to demonstrate eligibility. In summary, the IMT's focus on sequential insertion and minimal hashing overhead makes it well-suited for the frequently-updated census tree, offering better update performance and smaller circuits for verification than a traditional SMT structure in this context.

2.4 Zero-knowledge proofs

Zero-knowledge (ZK) proofs are cryptographic protocols that allow one party (the prover) to convince another (the verifier) that a certain statement is true, without revealing any information beyond the validity of the statement itself. In the context of voting, this enables participants to prove that their encrypted ballot is well-formed and complies with the election rules, without disclosing their actual choice.

ZK-SNARKs. We focus on a specific type of proof system called zero-knowledge succinct non-interactive arguments of knowledge (ZK-SNARKs). Informally, while a ZK proof convinces the verifier that a valid witness exists, an argument of knowledge additionally ensures that the prover actually knows such a witness, except with negligible probability. ZK-SNARKs extend this notion with two crucial properties: they are non-interactive, requiring only a single message from prover to verifier, and succinct, meaning that proof size and verification cost remain small regardless of the size of the underlying statement. For instance, Groth16 proofs [14], which rely on pairing-based cryptography over elliptic curves, are only about 200 bytes long and can be verified in a few milliseconds. These properties make ZK-SNARKs particularly well-suited for blockchain applications, where verification cost and data size are critical. In DAVINCI, ZK-SNARKs are used at multiple stages of the protocol: voters generate proofs to show that their encrypted ballots are correct and sequencers use recursive ZK-SNARKs to allow multiple proofs to be aggregated efficiently and generate proofs to attest to valid state transitions. In essence, ZK-SNARKs enable both voters and sequencers to produce compact proofs that can be checked on-chain, ensuring that all protocol rules are followed without requiring trust in any party.

Trusted setup. A limitation of some ZK-SNARK protocols is their reliance on a common reference string (CRS) generated during a so-called trusted setup. The CRS is built from random values that must not be known to either the prover or the verifier. To achieve this, the CRS can be created by a trusted third party or, more robustly, via a secure multi-party computation (MPC) protocol [10]. In an MPC setup, it suffices that at least one participant discards their secret contribution to ensure the integrity of the whole ceremony. For DAVINCI, we run an MPC for each of the circuits, see Section 7.2 and Section A.8.

Arithmetic circuits. In general, ZK-SNARKs operate by proving the satisfiability of an arithmetic circuit. An arithmetic circuit (from now on also called circuit or ZK circuit) is a directed acyclic graph where nodes represent addition or multiplication gates over a finite field. The inputs and outputs of the circuit correspond to the public inputs and private witnesses of the computation, while the intermediate wires carry intermediate values. Any deterministic computation, from verifying an encryption to checking Merkle proofs, can be compiled into an arithmetic circuit [12]. In practice, proving with ZK-SNARKs means showing knowledge of a secret witness (e.g., a plaintext ballot or encryption randomness) that satisfies all the equations encoded in the circuit. For example, the ballot circuit in DAVINCI encodes the constraints that a ciphertext

is well-formed, that the vote identifier is correctly derived, and that the ballot complies with the rules of the voting mode. The prover generates a proof of satisfiability, while the verifier only checks the proof against the public inputs, without learning the secret witness. By building complex protocol logic from arithmetic circuits, and using ZK-SNARKs to prove their satisfiability, DAVINCI ensures that every step of the election process, from ballot submission to state transitions, is enforced cryptographically and publicly verifiable.

3 Protocol intuition

The following section provides an overview of the DAVINCI protocol, describing its main phases and actors without delving into technical details. An election is governed by smart contracts deployed on Ethereum, which ensure transparency, enforce rules, and verify proofs. As shown in Fig. 1, the protocol consists of five main phases, from the setup of the election to the publication and verification of the final results.

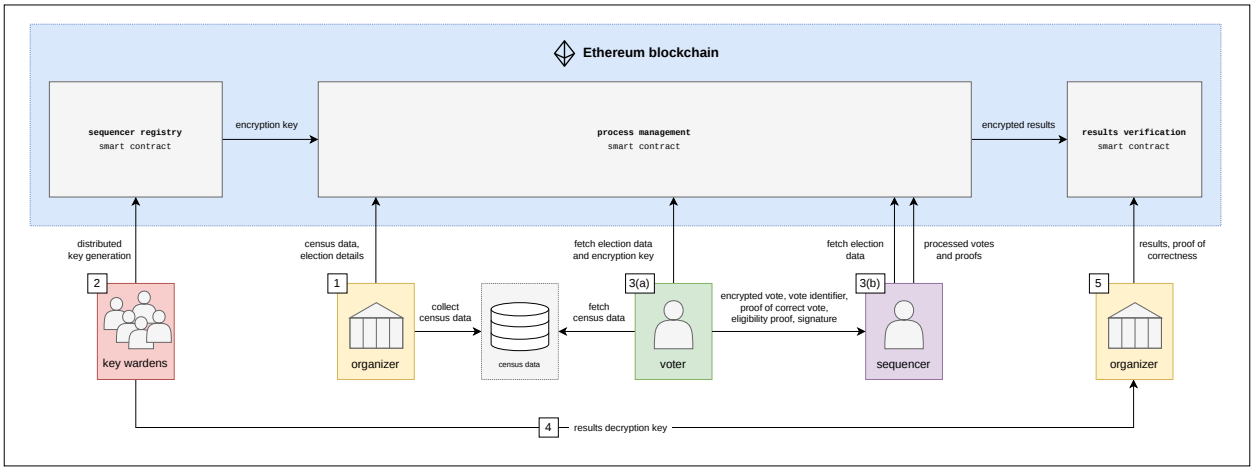

Fig. 1: Overview of the protocol flow.

1. *Election setup.* In the first phase, the *organizer* 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 *key wardens*, 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. Once the encryption public key is securely computed and published in the smart contract, the voting period can start.

3. *Voting period.* During the voting period, two actors interact continuously: *voters*, who cast their encrypted ballots, and *sequencers*, independent entities responsible for collecting, verifying, and processing these ballots before committing them to the shared state. Both processes, (a) ballot submission by voters and (b) vote batching by sequencers, occur continuously and in parallel throughout this phase. The voting period remains open until the deadline defined by the organizer, allowing voters to submit or overwrite ballots at any time, while sequencers periodically process new submissions and update the election state accordingly.

(a) *Vote casting.* Voters select their preferred choices according to the voting 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. Additionally, voters compute a unique vote identifier that will allow them to verify that their vote has been included in the final result. Alongside the encrypted vote and the vote identifier, 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. To mitigate coercion and vote-buying, the protocol allows voters to overwrite their vote any number of times during the voting phase.

(b) *Vote batching.* 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 election voting 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 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, 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.

4. *Tally decryption.* At the end of the voting period, the smart contract ceases to accept new state updates. Then, a threshold number of key wardens cooperate to reconstruct a partial decryption key that allows to decrypt the final tally. That is, instead of reconstructing the election's private key, each key warden publishes a partial decryption together with a proof that the contribution is correct. This way, the individual votes remain encrypted throughout the voting period, and only the final aggregated result is decrypted at the end of the election.

5. *Election finalization.* The results and a proof of correctness are submitted to the results verification smart contract. The contract verifies the proof, ensuring that the tally has been derived faithfully from the final state. Once this verification succeeds, the results and the latest state root remain available on-chain, providing an immutable record of the election outcome that can be independently audited by anyone.

4 Voting protocol

This section outlines the DAVINCI voting protocol. First, we introduce the parties involved in the process in Section 4.1. In Section 4.2 we present the Ethereum smart contracts that rule the DAVINCI protocol. Then, in Section 4.3, we describe the census data structures and in Section 4.4 the state Merkle trees. In Section 4.5, we present the ZK circuits used to enforce vote validity, authentication, aggregation, state transitions, and results computation. Finally, in Section 4.6, we detail the full protocol flow step by step, from the voting setup to the results validation and process finalization.

4.1 Parties involved

An election involves four types of participants: organizer, key wardens, voters, and sequencers.

Organizer. The organizer is the entity responsible for defining and setting up the election. 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.

Key wardens. Key wardens are a set of decentralized parties that collaboratively generate a public encryption key that voters use to encrypt their votes. The corresponding secret key is secret-shared among them so that a threshold of key wardens can later collaborate to decrypt the final tally.

Voters. Voters are the participants that belong the census and are allowed to cast their votes in the election.

TODO: we need to add that each voter has a secret sk and public pk key and an associated Ethereum address. Maybe add something about the weight as well? Consider adding a public-key scheme in Appendix A.

Sequencers. Sequencers are a set of parties that during the voting period receive and verify encrypted ballots from voters, reencrypt votes to prevent coercion, and update the shared public state accordingly.

4.2 Smart contracts

The DAVINCI protocol operates on an Ethereum-compatible network, which serves as the source of truth for the election. By leveraging an Ethereum virtual machine (EVM) blockchain, all election transitions become immutable and publicly verifiable. To coordinate and validate each phase of the process, DAVINCI deploys a set of smart contracts that enforce protocol rules, verify ZK proofs, and maintain on-chain commitments such as state roots, encryption keys, and final results. Together, these contracts provide a trustless execution environment that guarantees the correct and transparent progression of the election without relying on any centralized authority. **TODO**: add methods and state variables.

4.2.1 Organization registry

The OrganizationRegistry smart contract is used to create and manage organizations within the system. Each organization is identified by the Ethereum address of its creator and is associated with a name and a URL pointing to its metadata. At this stage, the contract serves only as a registration mechanism, but it lays the foundation for future extensions, such as managing multiple elections under the same organization.

4.2.2 Census management

The CensusManagement smart contract xxx.

4.2.3 Key management

The KeyManagement smart contract coordinates the DKG process among the key wardens. It records their contributions, verifies proofs of correct participation, and ensures that a valid encryption public key is produced once the DKG round is complete.

4.2.4 Election (process) registry

The ElectionRegistry manages the lifecycle of each election, from its creation to the registration of final results. It maintains the election's current state and ensures that all transitions follow the protocol.

4.2.5 State transition verifier

The StateTransitionVerifier is responsible for validating the proofs that attest to correct updates of the election state. It stores the verification key corresponding to the state transition circuit (see Section 4.5.4) and checks that each submitted proof correctly links the previous and new state roots. Only transitions verified through this contract are accepted as valid updates by the process registry.

4.2.6 Results verifier

The ResultsVerifier contract validates the proof of correct tally computation. It contains the verification key for the results circuit (see Section 4.5.5) and ensures that the final results submitted by the organizer match the commitments from the final state. Once verified, the results and their proof are permanently recorded on-chain, providing a tamper-proof reference for the election outcome.

4.3 Census

The census defines the set of eligible voters and their associated voting weights. At its core, a census is simply a dataset that, for each voter, contains a unique identifier (typically an Ethereum address), a sequential index, a voting weight, and a means to produce a membership proof that can be verified inside a ZK circuit. DAVINCI supports two mechanisms for generating such proofs:

- *Credential-based membership proofs*: eligibility is certified by a digital signature issued by a credential service provider (CSP). The CSP assigns a unique index (`idx`) to each voter and includes it in the signed credential alongside the voter's identifier (`address`) and weight (`weight`).
- *Merkle-tree-based membership proofs*: the census is organized as an incremental Merkle tree (see Appendix A.3) and denoted by $\mathsf{MT}^{\texttt{census}}$. Each voter is assigned a unique sequential index (`idx`) in the tree, and their identifier and weight are stored in the corresponding leaf. Only the Merkle root is kept on-chain, and sequencers supply Merkle proofs to attest membership. Each voter is assigned a unique sequential index (`idx`) at the time of registration, and their identifier (`address`) and weight (`weight`) are stored as a leaf at position `idx` in the tree. A membership proof (`censusProof`) for voter i is a Merkle path attesting that the tuple $(\texttt{address}_i, \texttt{weight}_i)$ is stored at position `idx` under the committed census tree root. Sequencers supply these Merkle proofs to attest voters' membership to the census and the index `idx` is used to derive the voter's reserved storage slot in the state tree (see Section 4.4).

To support different census models, the ElectionRegistry smart contract stores the following three parameters:

- `censusOrigin`: the deployment model (off-chain static, off-chain dynamic, on-chain dynamic, or CSP).
- `censusURI`: a uniform resource identifier (URI) pointing to an external reference (URL, GraphQL endpoint, CDN path, IPFS/IPNS, etc.) from which sequencers can download or query the census data.
- `censusRoot`: the on-chain commitment used for membership verification. Its interpretation depends on the `censusOrigin`. In the CSP model, this parameter corresponds to the hash of the CSP's public key, and in the case of Merkle trees, it corresponds to a Merkle root (off-chain static/dynamic) or a census contract address (on-chain dynamic).

Organizers are free to construct the census from various data sources, such as private membership registries, self-sovereign identity credentials, or Ethereum-based tokens (e.g. ERC-20 or NFTs) whose balances define eligibility. This flexibility supports a broad range of use cases and governance scenarios. The supported census models are summarized in Table 1 and described in detail below.

`censusOrigin`		`censusRoot`	`censusURI`	Membership proof
Credential service provider (CSP)		Hash of the CSP's public key	Endpoint to obtain an eligibility credential	Digital signature verification
Off-chain	Static	Fixed Merkle root	Endpoint to obtain the census tree	Merkle proof verified against the fixed Merkle root
	Dynamic (add-only)	Latest Merkle root	Endpoint to obtain the latest version of the tree	Merkle proof verified against the latest Merkle root
On-chain	Dynamic (add-only)	CensusManagement contract address	Endpoint to obtain the latest version of the tree	Merkle proof verified against most recent Merkle roots

Table 1: Summary of census configurations supported by the DAVINCI protocol.

Credential service provider. Voter eligibility is certified by a digital signature issued by a CSP. A voter obtains a signed credential from the CSP, and sequencers verify the signature inside the ZK circuits. The credential is issued over the tuple (`processID`, `idx`, `address`, `weight`) where `idx` is a unique index assigned by the CSP to each voter. This index plays the same role as in the Merkle-tree census models: it is used by the state transition circuit to derive the voter's reserved storage slot in the state tree (see Section 4.4). In

this case, `censusRoot` stores the hash of the CSP's public key, and `censusURI` specifies the endpoint where voters obtain their credential. This model is suitable when eligibility is managed externally through attested identities rather than by maintaining a Merkle-tree dataset.

Off-chain static census. The organizer constructs a fixed census prior to the election. The Merkle tree is built locally and the resulting root is stored in `censusRoot`, and `censusURI` specifies the endpoint from which sequencers can retrieve the full tree. Since the census is static, no updates occur during the voting period.

Off-chain dynamic census. The Merkle tree remains off-chain, but the organizer may append new voters during the voting period (never deleting voters or modifying weights, as this would enable double voting). Each new insertion produces a new Merkle root, which the organizer must publish to the ElectionRegistry contract by updating the `censusRoot` parameter. As in the static model, `censusURI` provides the endpoint from which sequencers can download the current version of the tree. Sequencers verify membership proofs against the most recently published root.

On-chain dynamic census. In this configuration, the census is managed by a dedicated on-chain contract (CensusManagement) that supports adding new members during the voting period. The address of this contract is stored in `censusRoot`, while `censusURI` points to an external representation of the tree. The CensusManagement contract maintains a history of valid Merkle roots. Sequencers may therefore generate membership proofs using older roots, and during state transitions the ElectionRegistry contract queries the CensusManagement contract to verify that the root used in the proof is still valid. As in all dynamic models, only additions to the census are permitted during the voting period.

4.4 State tree

The state Merkle tree, denoted by $\mathsf{MT}^{\mathtt{state}}$, represents the evolving global state of an election in DAVINCI. It is implemented as a SMT of fixed depth $D = 64$, as described in Appendix A.3. The state tree acts as a cryptographic accumulator that compactly commits to all relevant election data, including configuration parameters, vote identifiers, and encrypted ballots. By organizing the election state in a single SMT, DAVINCI supports efficient membership and update proofs that can be verified inside ZK circuits, enabling succinct and verifiable state transitions.

The root of the state tree, denoted `stateRoot`, is committed on-chain and serves as the authoritative reference to the current election state. Each state transition performed by a sequencer updates the tree and produces a new root. The correctness of this update, namely, that it follows the protocol rules and correctly incorporates new votes, is attested by a ZK-SNARK proof, which is verified by the StateTransitionVerifier smart contract before the new root is accepted.

Namespaced key space. The state tree operates over keys in the range $[0, 2^D - 1] \subset \mathbb{F}_p$ and uses a SNARK-friendly hash function (Poseidon, see Appendix A.2) to enable efficient in-circuit verification. To prevent collisions between different categories of data stored in the state, the 64-bit key space is partitioned into three disjoint numeric namespaces using a fixed thershold N. We define $N = 2^{D-1}$, which we use to split the address space into a lower region for configuration parameters and encrypted ballot storage, and an upper region strictly reserved for vote identifiers. This strict separation ensures that configuration entries, vote identifiers, and ballots never collide. The structure of the tree is depicted in Fig. 2.

Configuration namespace. The lowest indices of the tree are reserved for fixed process parameters that define the election rules. These keys are deterministic and occupy a negligible portion of the address space. Typical entries include:

- `0x0`: the election identifier (`processID`). [4]
- `0x2`: the ballot mode configuration, which is the set of rules for validating votes (`ballotMode`).
- `0x3`: the encryption public key used to encrypt the ballots (`encryptionKey`).

[4] The index `0x1` is reserved for backward compatibility (historically used for alternative census structures).

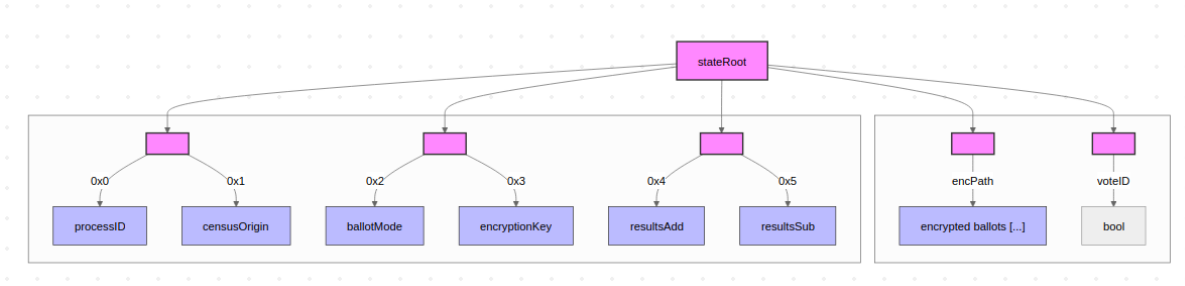

Fig. 2: Structure of the state Merkle tree. The first leaves are reserved for global election parameters while subsequent leaves store vote identifiers and encrypted ballots.

- `0x4`: an accumulator of the encrypted votes that need to be added to the tally (`resultsAdd`).
- `0x5`: an accumulator of the encrypted votes that have been overwritten (`resultsSub`).
- `0x6`: the census origin (`censusOrigin`).

Let `configMax` denote the largest index reserved for configuration parameters (by default, $\texttt{configMax} = 15$).

Encrypted ballot namespace. The region immediately following the configuration parameters, the interval $[\texttt{configMax} + 1, N - 1]$, is reserved for storing encrypted ballots. Ballot storage locations are deterministic and derived from the voter's position in the census. Let `idx` denote the voter's census index (see Section 4.3), proven via a census membership proof. The storage path for the encrypted ballot is computed as

$$\text{encPath} = \texttt{configMax} + \texttt{idx} \cdot 2^{16} + (\texttt{address} \bmod 2^{16}).$$

This construction assigns each eligible voter a unique, reserved leaf in the state tree. Because the index `idx` is authenticated by the census, a voter can only write to their own slot. This enables efficient *last-vote-wins* semantics: if a voter overwrites their ballot, the new ciphertext replaces the previous one at the same path, while the tally is updated by subtracting the old contribution and adding the new one.

Vote identifier namespace. The upper portion of the tree, $[N, 2^D - 1]$ is reserved for *vote identifiers* (`voteID`), which allows voters to verify that their vote has been included in the election state, without revealing or linking to the corresponding encrypted ballot. Vote identifiers are derived by hashing the process identifier, the voter's address, and fresh randomness, and then mapping the result into the allowed range by setting the most significant bit to 1:

$$\texttt{voteID} = N + \big(\text{Hash}(\texttt{processID}, \texttt{address}, k) \bmod N\big).$$

The StateTransitionCircuit recomputes this derivation and enforces that $N \leq \texttt{voteID} < 2^D$, guaranteeing that vote identifiers cannot overlap with configuration entries or ballot storage.

Unlike classical nullifiers, vote identifiers are not used to prevent duplicate voting directly. Instead, correctness is enforced at the circuit level: the state transition circuit guarantees that a vote identifier can only be inserted if the corresponding leaf in the state tree is empty. This emptiness check is enforced inside the ZK circuit. Since the vote identifier depends on a fresh random value k, if a collision occurs (i.e., the computed leaf is already occupied), the circuit will reject the transition. In that case, the voter must resample k and recompute a new vote identifier until an unused leaf is obtained. As a result, collisions do not compromise correctness or liveness, but merely require retrying the identifier derivation.

Note that the maximum number of vote identifiers that can be stored is bounded by the size of this namespace (2^{D-1}), which upper-bounds the total number of votes and overwrites that can be processed throughout the election. This capacity is orders of magnitude larger than any realistic election workload: historical elections peak at $\approx 10^9$ votes [19], ensuring that identifier exhaustion is practically impossible and that collisions are exceptionally rare [5].

[5] Based on the birthday paradox, the probability of at least one collision occurring is given by $P = 1 - e^{-n^2/2d}$, where n is the number of vote identifiers and $d = 2^{D-1}$. For a large-scale national election of 10^8 votes emitted, this

Together, the state tree provides a compact and tamper-proof commitment to the current status of the election. At the end of the election, the final state root together with the on-chain verification of all transitions serves as a complete cryptographic record of the election, encapsulating both the tally and the integrity of the entire voting procedure.

4.5 Circuits

At the core of DAVINCI lie a set of arithmetic circuits that enforce the correctness of every step of the election. Each circuit corresponds to a distinct task and, taken together, they guarantee that all protocol rules are satisfied without revealing any private information. Specifically, the *ballot circuit* (Section 4.5.1) ensures that encrypted votes are valid and well-formed, the *verifier circuit* (Section 4.5.2) verifies the voter's proof; the *aggregation circuit* (Section 4.5.3) combines multiple authenticated votes into a single proof; the *state transition circuit* (Section 4.5.4) updates the global state root and produces the ZK-SNARK proof that is verified on-chain; and the *results circuit* (Section 4.5.5) proves that the election results were computed correctly. The flow of these circuits and their interactions is shown in Fig. 3.

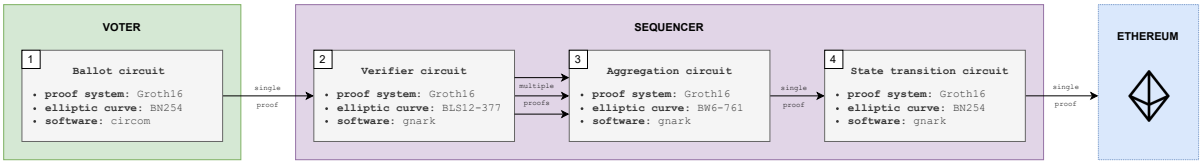

Fig. 3: Overview of the circuit flow in DAVINCI. *Results circuit* is missing.

Remark 1. For clarity, the circuits are described in this section with their full set of public inputs explicitly listed. In the actual implementation, however, we apply an optimization: rather than exposing all public inputs PI individually, we compute a single commitment $h = \mathsf{Poseidon}(\mathsf{PI})$ and use h as the only public input. The proof verifier then checks that $\mathsf{Poseidon}(\mathsf{PI}) = h$, ensuring that the original public inputs are consistent while reducing both proof size and verification cost. Note that this optimization does not alter the semantics of the circuits but significantly reduces verifier complexity and on-chain verification costs. Other implementation aspects such as the exact number of constraints and the proving frameworks used are deferred to Section 7.2.

4.5.1 Ballot circuit

The ballot circuit ($\mathsf{BallotCircuit}$), illustrated in Fig. 4, is generated locally by the voter at the time of casting a ballot. Its purpose is twofold: first, to prove that the encrypted ballot is valid and complies with the protocol rules defined by the organizer; and second, to prove that the vote identifier (`voteID`) has been correctly derived. The correctness of this circuit is attested by a ZK proof generated using the ZK-SNARK protocol described in Appendix A.8, instantiated over the BN254 curve. We call the resulting proof *ballot proof* and denote it as `ballotProof`. We detail below the inputs and constraints of the ballot circuit.

Public inputs

- public `processID`: election identifier.
- public `ballotMode`: ballot protocol configuration.
- public `encryptionKey`: encryption public key.
- public `address`: voter's address.

probability is vanishingly small ($\approx 0.054\%$), and for an extreme workload of 10^9 votes, the probability of a single collision is only $\approx 5.4\%$, meaning that the need to resample k remains highly improbable.

- public `weight`: voter's weight.
- public `encryptedBallot`: encrypted ballot.
- public `voteID`: vote's unique identifier.

Private inputs

- private `ballot`: voter's voting choice.
- private k: random scalar.

Constraints

- *Vote encryption.* Correct encryption of the ballot:

$$\texttt{encryptedBallot} = \mathsf{Enc}_{\texttt{encryptionKey}}(\texttt{ballot}; k).$$

- *Protocol rules compliance.* The voter's ballot meets the protocol rules (see Section 5):

$$\text{protocolRulesCompliance}(\texttt{ballotMode}, \texttt{weight}, \texttt{ballot}) = \texttt{true}.$$

- *Vote identifier.* The vote's identifier is correctly computed:

$$\texttt{voteID} = \text{Hash}(\texttt{processID} \,||\, \texttt{address} \,||\, k).$$

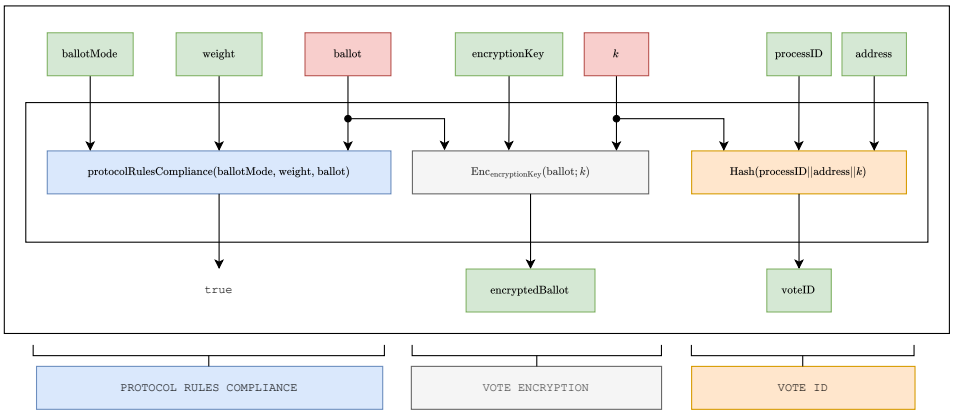

Fig. 4: Ballot circuit. All public values are framed in green.

4.5.2 Verifier circuit

The verifier circuit (VerifierCircuit), illustrated in Fig. 5, is generated by the sequencer when processing a vote. This circuit verifies the ballot's proof generated by the voter and the correctness of their digital signature. The correctness of this circuit is attested by a ZK proof generated using the ZK-SNARK protocol described in Section A.8, instantiated over the BLS12-377 curve. We call the resulting proof the *authentication proof* and denote it as `authenticationProof`. We detail below the inputs and constraints of this circuit. – Authentication of verification proof?

Public inputs

- public `processID`: election identifier.

- public `ballotMode`: ballot protocol configuration.
- public `encryptionKey`: encryption public key.
- public `encryptedBallot`: encrypted ballot.
- public `voteID`: vote's unique identifier.
- public `address`: voter's address.
- public `weight`: voter's weight.

Private inputs

- private pk: voter's public key.
- private `signature`: voter's signature.
- private `ballotProof`: voter's ballot proof.

Constraints

- *Authentication + non-malleability.* The voter submitted and signed their vote.
  - The signature of the vote's identifier is valid for the given public key:

$$\mathsf{S.Verify}_{\mathsf{pk}}(\texttt{voteID}, \texttt{signature}) = \texttt{true}.$$

  - The address is correctly derived from the user's public key:

$$\texttt{address} = \mathsf{Keccak}(\mathsf{pk}).$$

- *Voter's proof verification.* The voter's submitted proof is valid for the inputs provided.
  - Set public inputs:

$$\mathsf{PI} = (\texttt{processID}, \texttt{ballotMode}, \texttt{encryptionKey}, \texttt{encryptedBallot}, \texttt{voteID}, \texttt{weight}, \texttt{address}).$$

  - Verify proof:

$$\mathsf{P.Verify}(\texttt{ballotProof}, \mathsf{PI}) = \texttt{true}.$$

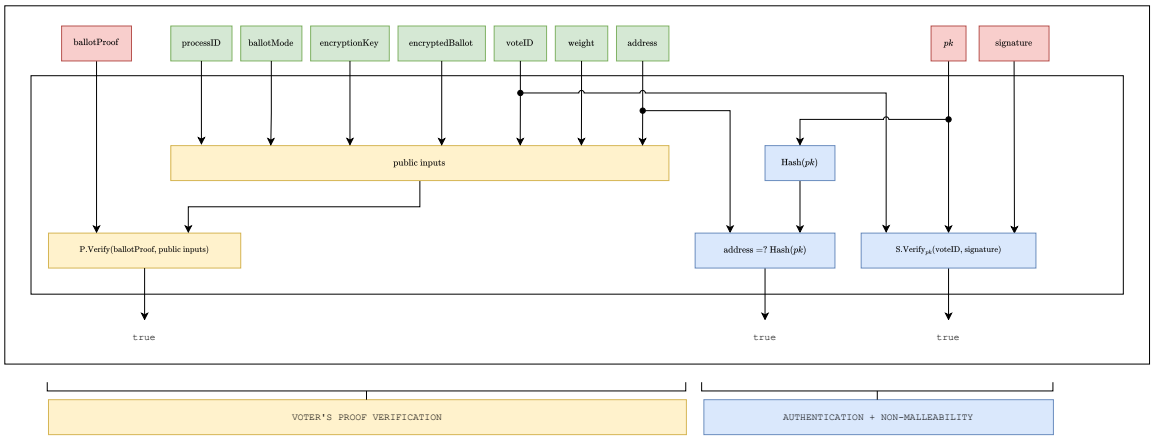

Fig. 5: Authentication circuit. All public values are framed in green.

4.5.3 Aggregation circuit

The aggregation circuit (AggregationCircuit), illustrated in Fig. 6, is generated by the sequencer to combine multiple authenticated votes into a single proof. Its purpose is to reduce verification overhead by recursively aggregating individual authentication proofs, while ensuring that all aggregated votes belong to the same election. The batch size (`batchSize`), i.e., the maximum number of proofs aggregated in a single execution, is a fixed parameter (currently set to 60). If fewer proofs are available, the sequencer pads the batch with dummy proofs so that the circuit always operates on a fixed-size input. The correctness of this circuit is attested by a ZK proof generated using the ZK-SNARK protocol described in Appendix A.8, instantiated over the BW6-761 curve. We call the resulting proof the *aggregation proof* and denote it as `aggregationProof`. We detail below the inputs and constraints of the aggregation circuit.

Public inputs

- public $\{\texttt{encryptedBallot}_i\}_{i=1}^{\texttt{batchSize}}$: set of voters' encrypted ballots.
- public $\{\texttt{voteID}_i\}_{i=1}^{\texttt{batchSize}}$: set of vote identifiers.
- public $\{\texttt{address}_i\}_{i=1}^{\texttt{batchSize}}$: set of voters' addresses.
- public `processID`: election identifier.
- public `ballotMode`: ballot protocol configuration.
- public `encryptionKey`: encryption public key.

Private inputs

- private $\{\texttt{authenticationProof}_i\}_{i=1}^{\texttt{batchSize}}$: set of authentication proofs.

Constraints

- *Votes aggregation.* The accumulated authentication proofs are valid. For every $i \in \{1, \ldots, \texttt{batchSize}\}$:
  - Set public inputs:

$$\mathsf{PI} = (\texttt{encryptedBallot}_i, \texttt{voteID}_i, \texttt{address}_i, \texttt{weight}_i, \texttt{processID}, \texttt{ballotMode}, \texttt{encryptionKey}).$$

  - Verify proof:

$$\mathsf{P.Verify}(\texttt{authenticationProof}_i, \mathsf{PI}) = \texttt{true}.$$

- *Shared public inputs.* All authentication proofs are verified using the same public inputs `processID`, `ballotMode`, and `encryptionKey`.

4.5.4 State transition circuit

The state transition circuit (StateTransitionCircuit), illustrated in Fig. 7, is generated by the sequencer to update the global state of the election. Its purpose is to verify that a batch of aggregated votes has been correctly incorporated into the state Merkle tree, that overwrites and vote identifiers are handled consistently, that the accumulators of encrypted results are updated accordingly, that each included vote corresponds to an eligible voter included in the census, and that the state transition data used by the sequencer matches the data made available through Ethereum data blobs. The correctness of this circuit is attested by a ZK proof generated using the ZK-SNARK protocol described in Appendix A.8, instantiated over the BN254 curve, which is supported by Ethereum's native precompiles for proof verification. We call the resulting proof the *state proof* and denote it as `stateProof`. It is submitted on-chain and verified by the process management smart contract, ensuring that every accepted state transition complies with the protocol rules.

To formalize the setting, let the sequencer collect a batch of N votes ($N \leq 60$), of which n are new vote submissions and $m = N - n$ are overwrites of previously cast ballots. Without loss of generality, we assume that the batch is ordered such that the first n votes correspond to new submissions, and the remaining m

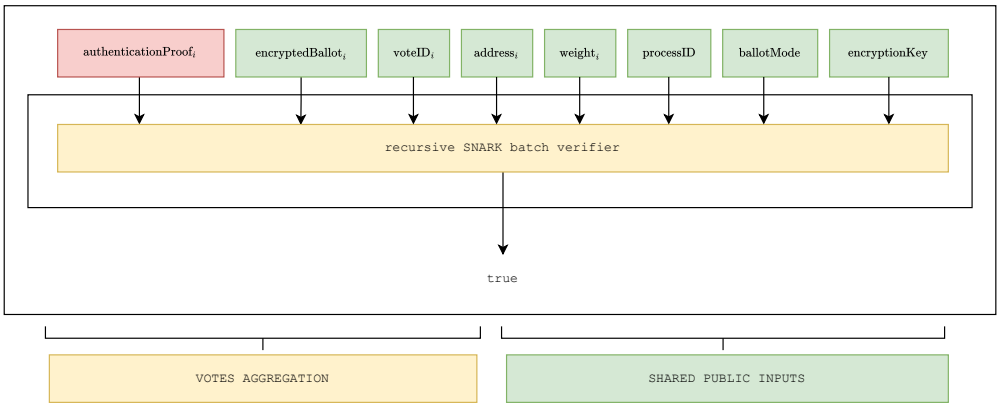

Fig. 6: Aggregation circuit. All public values are framed in green.

votes correspond to overwrites. In the following, we describe the inputs and constraints of the state transition circuit in detail.

Public inputs

- public `censusRoot`: Merkle root of the census tree.
- public $\texttt{stateRoot}^{\texttt{old}}$: current Merkle root of the state tree.
- public $\texttt{stateRoot}^{\texttt{new}}$: new tree root after all state transitions are applied.
- public $\texttt{numOverwrites}^{\texttt{old}}$: current number of overwrites.
- public $\texttt{numOverwrites}^{\texttt{new}}$: number of overwrites after all state transitions are applied.
- public $\texttt{numVotes}^{\texttt{old}}$: current number of votes.
- public $\texttt{numVotes}^{\texttt{new}}$: total number of votes after all state transitions are applied.
- public C: commitment to the data blob.

Private inputs

- private `processID`: election identifier.
- private `censusOrigin`: census origin parameter.
- private `ballotMode`: ballot protocol configuration.
- private `encryptionKey`: encryption public key.
- private $\texttt{resultsAdd}^{\texttt{old}}$: current accumulator of votes (before transitions are applied).
- private $\texttt{resultsSub}^{\texttt{old}}$: current accumulator of votes that have been overwritten (before transitions).

- private $\{\texttt{aggregationProof}_i\}_{i=1}^{N}$: set of aggregation proofs.
- private $\{\texttt{censusProof}_i\}_{i=1}^{N}$: census membership proofs.
- private $\{\texttt{voteID}_i\}_{i=1}^{N}$: set of vote's identifiers.
- private $\{\texttt{encryptedBallot}_i\}_{i=1}^{N}$: set of voters' encrypted ballots.
- private $\{\texttt{encryptedBallot}_i^{\texttt{old}}\}_{i=n+1}^{N}$: set of all old encrypted ballots that will be overwritten.
- private $\{\texttt{address}_i\}_{i=1}^{N}$: list of voters' addresses.
- private $\{\texttt{weight}_i\}_{i=1}^{N}$: list of voters' weights.

- private $\texttt{merkleInclusionProof}^{\texttt{processID}}$: proof that `processID` is stored in leaf `0x0` of the state tree.

- private $\texttt{merkleInclusionProof}^{\texttt{censusOrigin}}$: proof that `censusOrigin` is stored in leaf `0x1` of the tree.
- private $\texttt{merkleInclusionProof}^{\texttt{ballotMode}}$: proof that `ballotMode` is stored in leaf `0x2` of the state tree.
- private $\texttt{merkleInclusionProof}^{\texttt{resultsAdd}}$: proof that $\texttt{resultsAdd}^{\texttt{old}}$ is stored in leaf `0x3` of the state tree.
- private $\texttt{merkleInclusionProof}^{\texttt{resultsSub}}$: proof that $\texttt{resultsSub}^{\texttt{old}}$ is stored in leaf `0x4` of the state tree.

- private $\{\texttt{merkleNonInclusionProof}_i^{\texttt{encryptedBallot}}\}_{i=1}^{n}$: proof that the leaf corresponding to a voter's $\texttt{address}_i$ that submits a new vote is empty.
- private $\{\texttt{merkleInclusionProof}_i^{\texttt{encryptedBallot}}\}_{i=n+1}^{N}$: proof that the vote $\texttt{encryptedBallot}_i^{\texttt{old}}$ is stored in the leaf corresponding to a voter's `address` that submits the vote overwrite $\texttt{encryptedBallot}_i$.
- private $\{\texttt{merkleNonInclusionProof}_i^{\texttt{voteID}}\}_{i=1}^{N}$: non-membership proofs of vote identifiers.

- private k: random scalar.
- private openingProof: KZG opening proof.

Constraints

- *Aggregation verification.* The accumulated authentication proofs are valid. For every $i \in \{1, \dots, N\}$:
  - Set public inputs:

    $$\mathsf{PI} = (\texttt{weight}_i, \texttt{address}_i, \texttt{encryptedBallot}_i, \texttt{voteID}_i, \texttt{ballotMode}, \texttt{encryptionKey}, \texttt{processID}).$$

  - Verify proof:

    $$\mathsf{P.Verify}(\texttt{aggregationProof}_i, \mathsf{PI}) = \texttt{true}.$$

- *Census membership.* Each processed vote must correspond to an eligible voter, according to the census configuration specified by the parameter `censusOrigin`.
  - If `censusOrigin` = CSP, then verify a signature issued by the CSP. Concretely, for all $i \in \{1, \dots, N\}$:

    $$\mathsf{S.Verify}_{\texttt{censusRoot}}\big((\texttt{address}_i, \texttt{weight}_i), \texttt{censusProof}_i\big) = \texttt{true},$$

    where `censusRoot` contains the CSP public key and $\texttt{censusProof}_i$ is the eligibility signature attesting that voter i belongs to the census with the given weight. Missing parameters in the message being signed, e.g. ProcessID.
  - Otherwise, if the census is represented by an IMT, the circuit verifies a Merkle membership proof showing that the pair $(\texttt{address}_i, \texttt{weight}_i)$ is included in the committed census. For all $i \in \{1, \dots, N\}$:

    $$\mathsf{MT}^{\texttt{census}}.\mathsf{Verify}\big((\texttt{address}_i, \texttt{weight}_i), \texttt{censusProof}_i, \texttt{censusRoot}\big) = \texttt{true}.$$

- *Transition verification.* Verify that the initial state and the state transition updates are applied correctly.

- Verify that the inputs $\texttt{processID}$, $\texttt{ballotMode}$, $\texttt{encryptionKey}$, $\texttt{resultsAdd}^{\texttt{old}}$, $\texttt{resultsSub}^{\texttt{old}}$, and $\texttt{censusOrigin}$ correspond to the content of the first state tree leaves.

$$\mathsf{MT}^{\texttt{state}}.\mathsf{Verify}(\texttt{processID}, \texttt{0x0}, \texttt{merkleInclusionProof}^{\texttt{processID}}, \texttt{stateRoot}^{\texttt{old}}).$$
$$\mathsf{MT}^{\texttt{state}}.\mathsf{Verify}(\texttt{ballotMode}, \texttt{0x2}, \texttt{merkleInclusionProof}^{\texttt{ballotMode}}, \texttt{stateRoot}^{\texttt{old}}).$$
$$\mathsf{MT}^{\texttt{state}}.\mathsf{Verify}(\texttt{encryptionKey}, \texttt{0x3}, \texttt{merkleInclusionProof}^{\texttt{encryptionKey}}, \texttt{stateRoot}^{\texttt{old}}).$$
$$\mathsf{MT}^{\texttt{state}}.\mathsf{Verify}(\texttt{resultsAdd}^{\texttt{old}}, \texttt{0x4}, \texttt{merkleInclusionProof}^{\texttt{resultsAdd}}, \texttt{stateRoot}^{\texttt{old}}).$$
$$\mathsf{MT}^{\texttt{state}}.\mathsf{Verify}(\texttt{resultsSub}^{\texttt{old}}, \texttt{0x5}, \texttt{merkleInclusionProof}^{\texttt{resultsSub}}, \texttt{stateRoot}^{\texttt{old}}).$$
$$\mathsf{MT}^{\texttt{state}}.\mathsf{Verify}(\texttt{censusOrigin}, \texttt{0x6}, \texttt{merkleInclusionProof}^{\texttt{censusOrigin}}, \texttt{stateRoot}^{\texttt{old}}).$$

- Verify that none of the $\texttt{voteID}_i$ for all $i \in \{1, \dots, N\}$ are initially part of the tree:

$$\mathsf{MT}^{\texttt{state}}.\mathsf{Verify}(\texttt{voteID}_i, \texttt{merkleNonInclusionProof}_i^{\texttt{voteID}}, \texttt{stateRoot}^{\texttt{old}}).$$

- Ensure that new votes are not part of the tree yet. For $i = 1, \dots, n$:

$$\mathsf{MT}^{\texttt{state}}.\mathsf{Verify}(\texttt{address}_i, \texttt{merkleNonInclusionProof}_i^{\texttt{encryptedBallot}}, \texttt{stateRoot}^{\texttt{old}}).$$

- Ensure that overwrites do correspond to votes that were already on the tree. For $i = 1, \dots, m$:

$$\mathsf{MT}^{\texttt{state}}.\mathsf{Verify}(\texttt{encryptedBallot}_i^{\texttt{old}}, \texttt{address}_i, \texttt{merkleInclusionProof}_i^{\texttt{encryptedBallot}}, \texttt{stateRoot}^{\texttt{old}}).$$

- State transitions.
  - For every new vote $i = 1, \dots, n$:
    1. Reencrypt the ballot: $\texttt{encryptedBallot}_i = \mathsf{ReEnc}_{\texttt{encryptionKey}}(\texttt{encryptedBallot}_i, k)$.
    2. Refresh reencryption randomness: $k = \mathsf{MiMC}(k)$.
    3. Add reencrypted ballot to the tree: $\mathsf{MT}^{\texttt{state}}.\mathsf{Insert}(\texttt{encryptedBallot}_i, \texttt{address}_i)$.
    4. Add vote identifier to the tree: $\mathsf{MT}^{\texttt{state}}.\mathsf{Insert}(\texttt{voteID}_i)$.
    5. Increase votes counter: $\texttt{numVotes} = \texttt{numVotes} + 1$.
    6. Aggregate the vote to the results accumulator: $\texttt{resultsAdd} = \texttt{resultsAdd} + \texttt{encryptedBallot}_i$.
  - For every overwritten vote $i = n + 1, \dots, N$:
    1. Refresh reencryption randomness: $k = \mathsf{MiMC}(k)$.
    2. Reencrypt the ballot: $\texttt{encryptedBallot}_i = \mathsf{ReEnc}_{\texttt{encryptionKey}}(\texttt{encryptedBallot}_i, k)$.
    3. Overwrite the encrypted ballot: $\mathsf{MT}^{\texttt{state}}.\mathsf{Update}(\texttt{encryptedBallot}, \texttt{address}_i)$.
    4. Add vote identifier: $\mathsf{MT}^{\texttt{state}}.\mathsf{Insert}(\texttt{voteID})$.
    5. Increase votes counter: $\texttt{numVotes} = \texttt{numVotes} + 1$.
    6. Increase overwrites counter: $\texttt{numOverwrites} = \texttt{numOverwrites} + 1$.
    7. Aggregate the vote to the results accumulator: $\texttt{resultsAdd} = \texttt{resultsAdd} + \texttt{encryptedBallot}_i$.
    8. Aggregate the old vote to the accumulator: $\texttt{resultsSub} = \texttt{resultsSub} + \texttt{encryptedBallot}_i^{\texttt{old}}$.
- After all previous state transitions are done, recompute new tree root:

$$\texttt{stateRoot}^{\texttt{new}} = \mathsf{MT}^{\texttt{state}}.\mathsf{Root}().$$

- Reencrypt a set of old leafs from the tree: first, very their Merkle tree inclusion proofs, then reencrypt them, and finally update each leaf's content.

- *Data blob.*
  - From all leaves that changed after the state transitions in the $\mathsf{MT}^{\texttt{state}}$ tree, construct blob B.
  - Derive polynomial $p(x)$ from B.
  - Compute evaluation point $z = \mathsf{Poseidon}(\texttt{processID}, C, \mathsf{openingProof})$.
  - Compute evaluation $y = p(z)$.
  - Verify $\mathsf{C.Verify}(z, y, C, \mathsf{openingProof}) = \texttt{true}$.

Remark 2. Splitting the sequencer computation into three circuits provides an important efficiency benefit. If the global `stateRoot` changes while a sequencer is preparing a proof, only the last circuit must be re-run to reflect the updated state, avoiding the need to recompute all previous proofs.

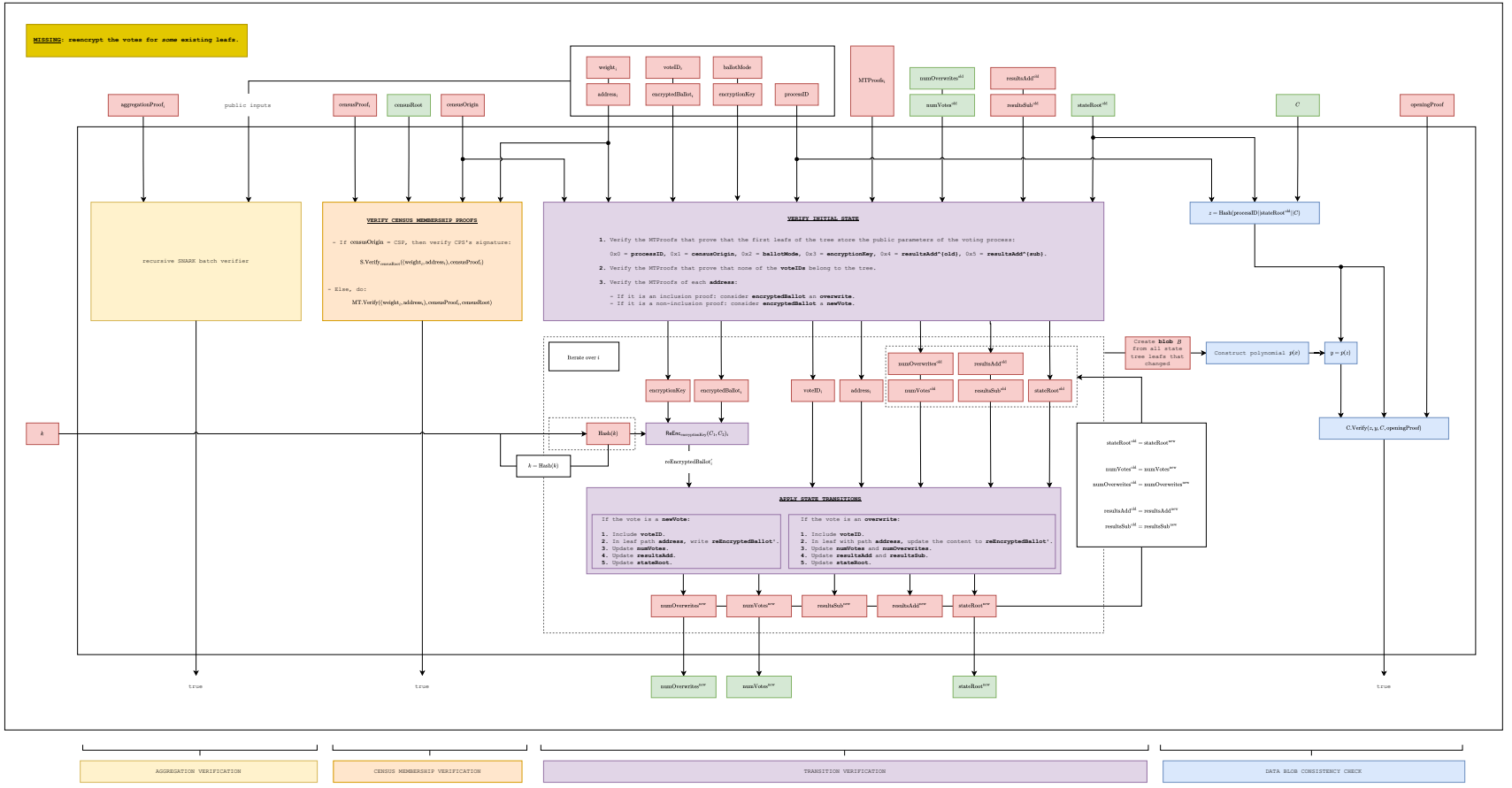

Fig. 7: State transition circuit. All public values are framed in green. **Missing inputs:** old encryptedBallots.

4.5.5 Results circuit

The results circuit (ResultsCircuit) proves that... .

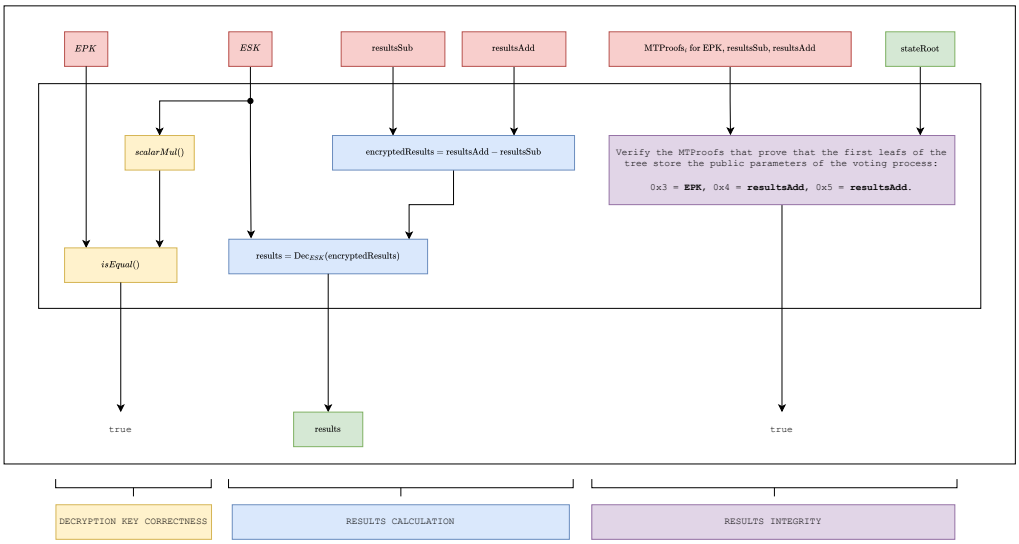

Fig. 8: Aggregation circuit. All public values are framed in green. Circuit is not correct.

4.6 Protocol flow

In this section we describe the overall flow of the DAVINCI voting protocol. An election is structured into five main phases: election setup, encryption key generation, voting period, tally decryption, and election finalization. Each phase specifies the interactions between the organizer, key wardens, voters, sequencers, and the smart contracts that coordinate the election. An overview of the complete process is shown in Fig. 9.

4.6.1 Election setup

Before any election can begin, sequencers must register by staking VOC tokens in the sequencer-registry smart contract (see Section 6). This registration is a one-time operation per sequencer and is independent of any particular election; only registered sequencers are eligible to participate in the DKG phase and to process votes. For each new election, the organizer performs the following steps:

1. Prepare the census data `censusData` and build the incremental Merkle tree $\mathsf{MT}^{\texttt{census}}$ from it. Publish the full tree to IPFS, and record the resulting URI as `censusURI`.
2. Compute the census commitment $\texttt{censusRoot} = \mathsf{MT}^{\texttt{census}}.\mathsf{Root}()$.
3. Define the election details, including duration, the census model `censusOrigin` (see Section 4.3), and the ballot rules `ballotMode` (see Section 5).
4. Upload the election metadata (questions, options, description, etc.) to IPFS to obtain electionMetadata.
5. Compute a unique process identifier `processID`, which is a 32-byte number derived from the organizer's Ethereum address, nonce, and the chain ID.
6. Initialize the state Merkle tree $\mathsf{MT}^{\texttt{state}}$ by inserting the election configuration parameters into the reserved configuration namespace (see Section 4.4): `processID` at leaf `0x0`, `ballotMode` at leaf `0x2`, `censusOrigin` at leaf `0x6`, and zero-valued accumulators `resultsAdd` and `resultsSub` at leaves `0x4` and `0x5`, respectively. Compute the initial state root – is this done by the SC?

$$\texttt{stateRoot} = \mathsf{MT}^{\texttt{state}}.\mathsf{Root}().$$

7. Set security parameters for the DKG ceremony (e.g. timeout, minimum number of sequencers) DKG.
8. Submit all this information in a transaction to the ElectionRegistry smart contract:

$$tx_{org} = (\texttt{processID},\ \texttt{censusRoot},\ \texttt{censusOrigin},\ \texttt{censusURI},\ \texttt{ballotMode},\ \textit{electionMetadata},\ \texttt{stateRoot},\ \textit{dkgParameters}).$$

Once this transaction is accepted, the election is publicly registered on-chain and the DKG phase can begin.

4.6.2 Encryption key generation

In this phase, key wardens collectively generate the public encryption key.

Key wardens fetch the process parameters (`processID`, `ballotMode`, `censusRoot`) from the ElectionRegistry smart contract and participate in a DKG protocol before the timeout set by the organizer expires. If the minimum number of contributions is reached, the public encryption key `encryptionKey` is derived and made available on-chain. This ensures that no single party controls the secret key, and that decryption later requires threshold participation (give more details).

Alternatively, generate a SNARK proving the EPK has been generated correctly and the contract verifies it.

4.6.3 Voting period

In this phase, voters cast their ballots and send them to the sequencers, who collect and process them.

(a) Vote casting. To cast a ballot, a voter does the following:

1. Select a sequencer.
2. Retrieve their census membership proof `censusProof` to prove eligibility.
3. Fetch parameters `encryptionKey`, `ballotMode`, and `processID` from the ElectionRegistry smart contract.
4. Select their ballot choice `ballot` according to the rules of `ballotMode`.
5. Sample fresh randomness $k \in \mathbb{F}$ and encrypt the ballot using `encryptionKey`:

$$\texttt{encryptedBallot} = \mathsf{Enc}_{\texttt{encryptionKey}}(\texttt{ballot}; k). \tag{1}$$

6. Use their Ethereum address `address` and the previous k to compute a unique vote identifier:

$$\texttt{voteID} = \mathsf{Poseidon}(\texttt{processID} \,||\, \texttt{address} \,||\, k). \tag{2}$$

7. Use the BallotCircuit from Section 4.5.1 to generate a ZK-SNARK proof to prove that Eqs. (1) and (2) are correctly computed and that `ballot` satisfies the ballot protocol rules according to `ballotMode`:

$$\texttt{voteProof} = \mathsf{P.Prove}(\mathsf{BallotCircuit}, \mathsf{witness} = (\texttt{ballot}, k), \mathsf{PI} = (\texttt{processID}, \texttt{ballotMode}, \texttt{encryptionKey}, \texttt{address}, \texttt{weight}, \texttt{encryptedBallot}, \texttt{voteID})).$$

8. Sign the vote identifier with their Ethereum secret key sk to authenticate the vote and ensure that was cast by a legitimate voter:

$$\texttt{signature} = \mathsf{S.Sign}_{\mathsf{sk}}(\texttt{voteID}).$$

9. Submit the package

$$\texttt{vote} = [\texttt{processID}, \texttt{voteID}, \texttt{encryptedBallot}, \texttt{censusProof}, \texttt{voteProof}, \texttt{signature}]$$

to the chosen sequencer.

(b) Vote batching. Upon receiving votes, a sequencer does the following:

1. Retrieve election identifier `processID`, current state root `stateRoot`, and census root `censusRoot` from the ElectionRegistry smart contract and the associated data from the Ethereum data blobs.

2. Upon receiving a vote

$$\texttt{vote}_i = [\texttt{processID}, \texttt{voteID}, \texttt{encryptedBallot}, \texttt{censusProof}, \texttt{voteProof}, \texttt{signature}],$$

extract the voter's public key $\mathsf{pk}_i = \mathsf{S.ExtractPublicKey}(\texttt{signature})$.

3. For each $\texttt{vote}_i$ received, use VerifierCircuit from Section 4.5.2 to generate a proof

$$\begin{aligned}\texttt{authenticationProof}_i = \mathsf{P.Prove}(&\mathsf{VerifierCircuit}, \mathsf{witness} = (\texttt{ballotProof}, \texttt{weight}, \texttt{censusProof}, \texttt{signature}),\\ &\mathsf{PI} = (\texttt{processID}, \texttt{ballotMode}, \texttt{encryptionKey}, \texttt{encryptedBallot}, \texttt{voteID},\\ &\texttt{censusRoot}, \mathsf{pk})).\end{aligned}$$

This proof ensures that the voter's voteProof, the signature, and the censusProof are all valid.

4. Batch a set of n proofs $\{\texttt{authenticationProof}_i\}_{i=1}^n$ and batch verify them together using AggregationCircuit from Section 4.5.3:

$$\begin{aligned}\texttt{aggregationProof} = \mathsf{P.Prove}(&\mathsf{AggregationCircuit}, \mathsf{witness} = (\{\texttt{authenticationProof}_i\}_{i=1}^n),\\ &\mathsf{PI} = (\texttt{encryptedBallot}, \{\texttt{voteID}_i\}_{i=1}^n), \{\mathsf{pk}_i\}_{i=1}^n, \texttt{processID}, \texttt{ballotMode},\\ &\texttt{encryptionKey}, \texttt{censusRoot}).\end{aligned}$$

This proof ensures that the public inputs corresponding to the global process parameters (processID, ballotMode, encryptionKey, censusRoot) are correct and that all $\texttt{authenticationProof}_i$ are valid.

5. Prepare the state transition data that will be stored in an Ethereum blob: pack votes/results into a 4096-cell blob blobData.
   - Compute data commitment $\mathsf{blobCommitment} = \mathsf{C.Commit}(\mathsf{blobData})$.
   - Derive evaluation point $\mathsf{evalPoint} = \mathsf{Poseidon}(\texttt{processID}, oldRoot, \mathsf{blobCommitment})$.
   - Prepare polynomial P from blobData.
   - Evaluate $y = P(\mathsf{evalPoint})$.
   - Generate opening proof openingProof from ....
   - The following step (circuit) will prove that $y = P(\mathsf{evalPoint})$.
6. Before doing state transitions, we need to prepare the data that will be stored on chain as a blob. Hence, collect xxxx, create commitment. In circuit (next step), we will prove that... .
7. Finally, prove that all transitions are correct .... with circuit from Section 4.5.4. consistency between data blobs and the on-chain state:

$$\texttt{stateProof} = \mathsf{P.Prove}(\mathsf{StateTransitionCircuit}, \mathsf{witness} = (), \mathsf{PI} = ()).$$

8. Verify each vote submission received by generating a state transition proof (see Section 4.5.4), ensuring:
   - correct accumulation of encrypted votes via ElGamal's homomorphic properties,
   - eligibility of voters (via census Merkle proofs),
   - absence of double voting (or correct handling of overwrites via nullifiers),
9. Submit the updated state root to the smart contract, while the full data are stored in Ethereum blobs.

Sequencers repeat this process until the voting deadline set by the organizer.
Upon receiving a valid transaction, the ElectionRegistry smart contract does the following checks:

4.6.4 Tally decryption

After the voting period expires, t out of n sequencers publish their decryption shares of the election private key. Once the threshold is reached, the election secret key can be reconstructed (on-chain or off-chain), enabling the decryption of the aggregated tally.

Rewards and penalties for sequencers are managed according to their correct participation: sequencers are rewarded proportionally to the number of votes sequenced, and may be slashed for failing to provide a valid decryption share (see Seciton 6 for details on incentivation mechanisms.)

4.6.5 Election finalization

In this final phase, the organizer/a sequencer computes the tally and publishes the result, together with a proof of its correct computation, on-chain.

1. Retrieve `decryptionKey` from ....
2. ...
3. Use ResultsCircuit from Section 4.5.5 to generate a proof

$$\texttt{resultsProof} = \mathsf{P.Prove}(\mathsf{ResultsCircuit}, \mathsf{witness} = (...), \mathsf{PI} = (...)).$$

The election is considered finalized once the decrypted tally is available on-chain. At this stage, both the election results and the complete integrity of the process are permanently recorded and publicly auditable.

Note that, anyone can verify the correctness of the final results by checking the ZK-SNARK state proofs and on-chain commitments. Moreover, the combination of the final state root, the decryption of the accumulators, and the publicly verifiable ZK-SNARK proofs guarantees the integrity of the entire election.

5 Ballot protocol

The ballot protocol provides a unified and parametric way to represent a wide range of voting systems within DAVINCI. Instead of designing a separate circuit for each voting rule, the protocol defines ballots as fixed-length arrays of integers, subject to a small set of configurable parameters. These parameters constrain the values that can appear in each field of the ballot and the aggregate properties of the ballot as a whole. By adjusting these parameters, the same circuit can implement approval voting, ranking, quadratic voting, multiple choice, and many other schemes. In the protocol we refer to the different configurations as the *ballot mode* (`ballotMode`). This abstraction has two key advantages. First, it allows a broad variety of voting systems to be supported with a minimal circuit design, avoiding conditional logic that would otherwise increase constraint counts. Second, it provides a common interface for tallying, since all ballots are aggregated into a single results array regardless of the voting mode.

Definition of parameters. Each ballot is defined as an array of integer values subject to a set of configurable parameters, illustrated in Fig. 10. These parameters are defined as follows:

- `numFields`: the maximum number of fields (i.e., options) in the ballot.
- `minValue`: the minimum value that each field in the ballot can take.
- `maxValue`: the maximum value that each field in the ballot can take.
- `uniqueValues`: a Boolean flag indicating whether all field values must be different (as in ranking systems).
- `costExponent`: the exponent applied when computing the cost of casting votes in a field. This parameter enables quadratic or higher-order voting rules.
- `minValueSum`: the minimum allowed total sum of a ballot, computed as $\sum_{i=1}^{\texttt{numFields}} v_i^{\texttt{costExponent}}$, where v_i are the field values.
- `maxValueSum`: the maximum allowed total sum of a ballot, computed as $\sum_{i=1}^{\texttt{numFields}} v_i^{\texttt{costExponent}}$, where v_i are the field values (e.g., to enforce a budget of credits).

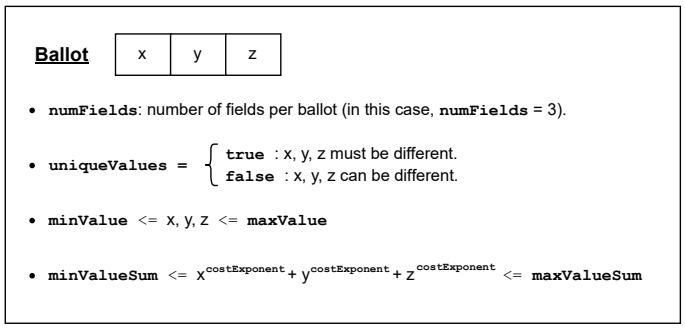

Fig. 10: Schematic representation of the parameters that define a ballot in DAVINCI.

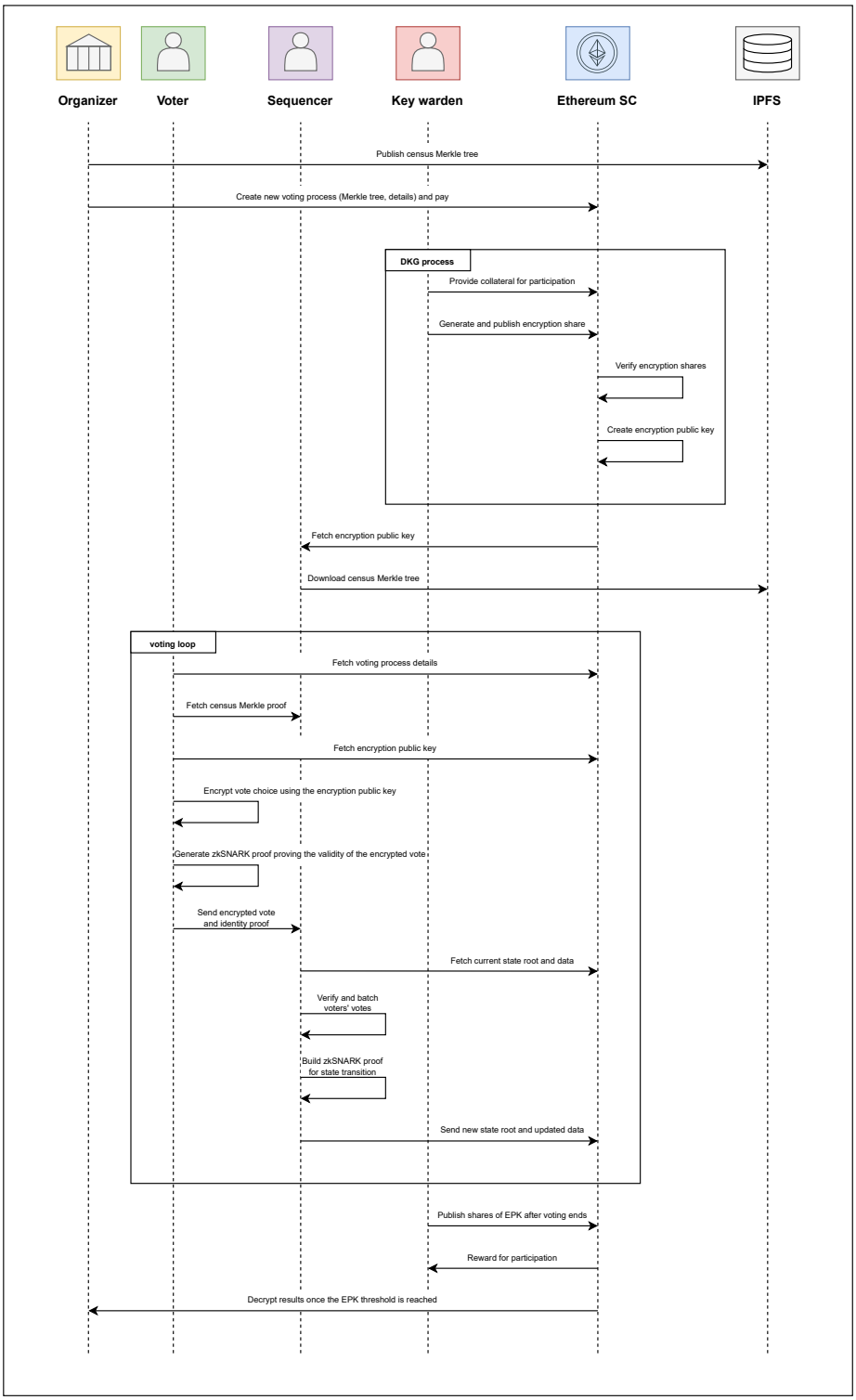

Fig. 9: Vocdoni voting process overview. Figure is OUTDATED.

Definition of constraints. Given a ballot with n fields and each field filled with a value v_i for $i = 1, \ldots, n$, we enforce the following constraints (captured in Fig. 10):

- $\texttt{numFields} = n$,
- If $\texttt{uniqueValues} = \texttt{true}$, then $v_i \neq v_j$ for all $i \neq j$.
- $\texttt{minValue} \leq v_i \leq \texttt{maxValue}$ for all $i \in \{1, \ldots, n\}$.
- $\texttt{minValueSum} \leq \sum_{i=1}^{n} v_i^{\texttt{costExponent}} \leq \texttt{maxValueSum}$.

General mechanism. A ballot is valid if and only if all the above constraints are satisfied. Invalid ballots are rejected at the circuit level and do not contribute to the tally. Valid ballots are aggregated into a results array, where each entry is the sum of all votes cast for the corresponding field across all voters. Voter weights, defined in the census tree (see Section 4.3), are taken into account when computing these sums. By default, all voters have the same weight, but the protocol also supports weighted voting, where different participants may contribute proportionally to their assigned voting power. In this case, `maxValueSum` reflects the maximum weight assigned to each voter, and the ballot and verifier circuits enforce that the correct weight is being used (see Sections 4.5.1 and 4.5.2 for further details).

Ballot modes. Ballot modes correspond to different voting systems, each adjusted with the variables above ballot configurations. For usability, the organizer does not need to set every parameter manually: instead, they simply select one of the predefined ballot modes (e.g., quadratic voting, ranking, approval), each corresponding to a fixed parameter configuration. Some representative modes are shown in Fig. 11, which illustrate the expressiveness of the ballot protocol. For example, in *approval voting*, each field is binary and voters can approve or reject multiple options simultaneously. In *ranking*, fields must form a permutation of the integers from 1 to N, enforcing uniqueness of values. In *quadratic voting*, voters allocate credits across options, and the cost is quadratic in the number of credits used, captured by setting $\texttt{costExponent} = 2$. Other systems such as single-choice or multiple-choice elections are special cases of the same framework. The figure also illustrates examples of voters' ballots, with the last column indicating whether each ballot is valid or invalid under the rules of the selected mode.

6 Incentive mechanisms

In this section, we describe/propose incentive mechanisms. However, this is not the only way. Additionally, we should change the word sequencer by key warden wherever applicable.

6.1 The Vodoni 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 (organizer, key wardens, voters, and sequencers), ensuring the integrity, efficiency, and security of the system. In particular, the token has the following roles.

- *Collateral for sequencers.* Sequencers must stake VOC tokens as collateral to participate. This serves as a safeguard to ensure responsible participation. Misbehavior or failure to meet obligations can result in penalties, including partial or total loss of the stake.
- *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.
- *Payment for elections.* Organizers pay fees in VOC tokens to create and manage elections. These fees depend on factors such as registry size, voting duration, and desired security level.
- *Governance.* Token holders can participate in the decentralized governance of the project, influencing protocol upgrades, ecosystem development, and other initiatives. This ensures that the project evolves in a transparent, community-driven manner.

Name	Description	Example	numFields	minValue	maxValue	uniqueValues	costExponent	minValueSum	maxValueSum	Ballots example							
											1	2	3	4	5	totalValueSum	validVote
Approval voting	Multiple fields with only a 0-1 option	A referendum with 5 yes/no questions	5	0	1	FALSE	1	0	5	ballot 1	0	1	0	1	1	3	1
										ballot 2	1	1	1	1	1	5	1
										ballot 3	0	1	0	0	0	1	1
										total	**1**	**3**	**1**	**2**	**2**		**3**
											1	2	3	4	5	totalValueSum	validVote
Rating	Any field can be rated independently of the others	Satisfaction survey on 5 items, rated from 0 to 10 stars	5	0	10	FALSE	1	0	50	ballot 1	8	6	4	7	10	35	1
										ballot 2	5	4	6	12	4	31	0
										ballot 3	0	1	3	5	2	11	1
										total	**8**	**7**	**7**	**12**	**12**		**2**
											1	2	3	4	5	totalValueSum	validVote
Ranking	Number all options from 1 to N, using each number exactly once	Rank 5 destinations for the end-of-year school trip	5	1	5	TRUE	1	6	15	ballot 1	1	3	2	4	5	15	1
										ballot 2	1	5	3	4	5	18	0
										ballot 3	0	1	3	5	2	11	0
										total	**1**	**3**	**2**	**4**	**5**		**1**
											1	2	3	4	5	totalValueSum	validVote
Quadratic	Distribute M credits among N options (quadratically)	You have 12 credits to distribute among 5 options	5	0	12	FALSE	2	0	12	ballot 1	1	1	2	0	0	6	1
										ballot 2	3	0	0	0	2	13	0
										ballot 3	3	1	0	1	0	11	1
										total	**4**	**2**	**2**	**1**	**0**		**2**
											1	2	3	4	5	totalValueSum	validVote
Single choice	Select 1 among N options	Select a single winner from among 5 candidates	5	0	1	FALSE	1	1	1	ballot 1	1	0	0	0	0	1	1
										ballot 2	0	1	0	0	0	1	1
										ballot 3	0	1	1	0	0	2	0
										total	**1**	**1**	**0**	**0**	**0**		**2**
											1	2	3	4	5	totalValueSum	validVote
Multiple choice	Select M among N options	Select up to 3 finalists from 5 candidates	5	0	1	FALSE	1	0	3	ballot 1	1	0	1	1	0	3	1
										ballot 2	0	1	1	0	1	3	1
										ballot 3	1	2	3	0	0	6	0
										total	**1**	**1**	**2**	**1**	**1**		**2**

Fig. 11: Table showing various voting models using different ballot configurations. Each row corresponds to a voting system specified by a fixed configuration of the ballot parameters (`numFields`, `minValue`, `maxValue`, `uniqueValues`, `costExponent`, `minValueSum`, `maxValueSum`). The table also includes example ballots for each mode, illustrating how the constraints are enforced in practice. The last column indicates whether a ballot is valid (1) or invalid (0) under the rules of the selected mode.

6.2 Economics for organizers

Organizers cover the costs of elections in VOC tokens. The total cost combines four components:

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

where

- $\mathsf{baseCost}$ is a fixed setup fee, independent of the election duration or security level. It is calculated as

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

  where $\mathsf{fixedCost}$ is a protocol-defined fee, $\mathsf{maxVotes}$ the maximum number of votes (votes or voters?), and p a linear factor. This portion is not reimbursable and always rewarded to sequencers.
- $\mathsf{capacityCost}$ accounts for limited sequencer capacity. That is, is models the cost of reserving space for voting events relative to the number of available sequencers, number of voting events running, and the maximum number of voters. Costs rise non-linearly as available capacity decreases as

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

  with k_1 a scaling factor, $\mathsf{totalVotingProcesses}$ the number of elections running, $\mathsf{totalSequencers}$ the number of registered sequencers, $\mathsf{usedSequencers}$ the number of sequencers handling other elections, ϵ a small constant to avoid division by zero, and a an exponent controlling non-linearity.
- $\mathsf{durationCost}$ grows with the length of the voting period, scaled non-linearly with the formula

  $$k_2 \cdot \mathsf{processDuration}^b,$$

  where $\mathsf{processDuration}$ is measured in hours, k_2 is a scaling factor, and b controls non-linear growth. Shorter elections are cost-efficient, while longer ones become increasingly expensive.

- securityCost models the number of sequencers used, growing exponentially with diminishing returns:

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

  where k_3 is a scaling factor, c controls the steepness, numSequencers is the number of sequencers needed in the election, totalSequencers the number of available sequencers, and d adjusts the non-linearity as the number of sequencers increases.
  Before totalSequencers was defined as the number of *registered* sequencers and here it means the number of *available* sequencers. Is it assumed to always be the same?

To avoid impractical scenarios, the following two constraints are enforced:

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

Reimbursements. Organizers initially reserve resources for all eligible voters, assuming maximum turnout. Since this rarely occurs, unused portions may be reimbursed. The reimbursement is defined as

$$\text{reimbursement} = \text{totalCost} - \text{totalReward} - \text{baseCost},$$

where totalReward is the actual amount distributed to sequencers based on their participation. This mechanism ensures organizers do not overpay for unused capacity, while sequencers are still compensated for committed resources.

6.3 Economics for sequencers

Sequencers must stake VOC in the sequencer registry smart contract to participate. Rewards are based on:

- The number of votes included in the shared state.
- The number of vote rewrites (either overwrites or re-encryptions). Rewrites enhance receipt-freeness and are incentivized, though limited by protocol constants.
- The ratio of processed to non-processed votes relative to the maximum allowed voters.

The reward function for the i-th sequencer is

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

subject to the constraints

$$\frac{\text{voteRewrites}_i}{\text{votes}_i} \leq T, \quad \text{totalReward} = R + W, \quad R > W,$$

where T is the maximum allowed ratio of rewrites to votes. Rewards prioritize new votes over rewrites, ensuring sequencers cannot maximize profits by simply re-encrypting existing ballots.

Penalties. Sequencers failing to provide required decryption shares or misbehaving face slashing penalties as

$$\text{slashedAmount}_i = s \cdot \text{stakedCollateral}_i,$$

where $0 \leq s \leq 1$ is a slashing coefficient. This ensures accountability and discourages free-riding.

6.4 Summary and remarks

The cost model combines four components: base, capacity, duration, and security. It ensures small elections are cost-efficient, large or resource-intensive elections incur higher costs, and impractical setups are excluded. Organizers aim to minimize costs, while sequencers seek to maximize rewards — a natural tension that can be modeled as a strategic game. Analyzing this equilibrium is left for future work. Move this to Section 6?

7 Analysis

The DAVINCI protocol was designed according to a set of guiding principles: cryptography is the sole source of truth; no single entity must be trusted; the system should be modular, open source, and resilient; and it should remain scalable, automated, and accessible to a wide range of users. Building on these principles, we now discuss the concrete security properties of the protocol, followed by implementation details and performance results. – This whole section is still [WIP].

→ check https://ethresear.ch/t/vocdoni-protocol-enabling-decentralized-voting-for-the-masses-with-zk-technology/21036.

7.1 Security discussion

Based on the above principles, the protocol provides a number of concrete security properties that ensure the integrity, confidentiality, and verifiability of voting events. In what follows we discuss how DAVINCI achieves these properties, with particular emphasis on receipt-freeness, privacy, unlinkability, and robustness against quantum threats.

Receipt-freeness. Receipt-freeness prevents voters from proving to others how they voted, thereby mitigating coercion and vote-buying. In DAVINCI this is achieved through re-encryption, ballot overwrites, and randomized state updates.

- *Ballot re-encryption*: since our encryption scheme supports re-randomization, that is, a ciphertext can be refreshed with new randomness without changing the underlying message, sequencers exploit this by re-encrypting the received ballots before committing them to the state Merkle tree, making it computationally infeasible to link a submitted ciphertext with the stored one (give details of what does *computationally infeasible* mean here).
- *Handling receipts*: because ballots are re-randomized, voters cannot generate a receipt by revealing the randomness r used in encryption, since the ciphertext on-chain no longer corresponds to their r. This blocks vote-selling and coercion.
- *Overwrites*: voters may cast a new ballot at any time, replacing their earlier submission. This ensures that even if coercion occurs, a voter can subsequently change their vote as many times as desired, preserving their ability to express their true choice. When an overwrite occurs, the sequencer substracts the previous ballot from the subtractive accumulator, adds the new ballot to the additive accumulator, and re-encrypts the updated ballot before storing it in the state tree.
- *Concealing overwrites*: to prevent observers from distinguishing overwrites from routine re-randomizations, sequencers periodically re-encrypt a random subset of stored ballots. This obfuscation ensures that overwrites remain indistinguishable from normal re-randomizations, strengthening receipt-freeness.

Privacy. Ballots remain secret even though encrypted ballots are publicly stored and processed. Ballot secrecy is preserved through ElGamal encryption, which allows votes to be aggregated without decryption. Encrypted ballots are stored in public repositories (e.g., Ethereum blobs). Because of DKG sequencers cannot decrypt individual ballots themselves. The protocol does not reveal the ballot but it does reveal if someone has voted or not. Individual ballots remain private because the protocol only invokes threshold decryption on the final aggregate, not on each ballot separately.

Quantum resistance. Quantum computers threaten discrete-logarithm-based cryptography such as ECDSA and ElGamal. For this reason, DAVINCI is designed in a modular way that would allow to migrate to post-quantum primitives in the longer term. For example,use CRYSTALS-Dilithium, Falcon, or Rainbow, for signature schemes, Brakerski-Gentry-Vaikuntanathan (BGV) or Brakerski/Fan-Vercauteren (BFV) which are lattice-based homomorphic encryption schemes, and ZK-STARKs for post-quantum zero-knowledge proofs. Add citations to protocols.

Data availability. Ballots and state updates are stored in Ethereum blobs. Since these blobs may eventually be pruned from the blockchain, ensuring long-term data availability is an open problem. Possible solutions include decentralized storage networks or dedicated data-availability layers. This remains an active area of research.

End-to-end verifiability. Voters can check that their own ballot was included correctly, while anyone can audit the entire process to verify the final tally. 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.

7.2 Implementation

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. 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. 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. 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. Add link to repositories and details of the software used.

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.

MPC for CRS. Explain the trusted setup ceremony.

7.3 Performance evaluation

Work in progress.

8 Conclusions

9 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 (Iden3 and Polygon), Adrià Massanet (Privacy Scaling Explorations, Ethereum Foundation), Arnaucube (0xPARC), Javier Herranz (Polytechnic University of Catalonia), Jordi Puiggali (Secrets Vault), and Carla Ràfols (Pompeu Fabra University).

References

1. AAAATODOauthor: TODOtitle (9999), TODOurl
2. 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)
3. Baylina, J., Bellés, M.: Sparse Merkle trees (2019), https://docs.iden3.io/publications/pdfs/Merkle-Tree.pdf
4. 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
5. Bertoni, G., Daemen, J., Peeters, M., Van Assche, G.: The KECCAK SHA-3 submission (2011), https://keccak.team/files/Keccak-submission-3.pdf
6. Blake, I.F., Seroussi, G., Smart, N.P.: Elliptic curves in cryptography. Cambridge University Press, USA (1999)
7. 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
8. 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
9. Buterin, V., Feist, D., Loerakker, D., Kadianakis, G., Garnett, M., Taiwo, M., Dietrichs, A.: EIP-4844: Shard blob transactions, Ethereum Improvement Proposals, no. 4844, [online serial] (February 2022), https://eips.ethereum.org/EIPS/eip-4844
10. Canetti, R.: Universally composable security: a new paradigm for cryptographic protocols. In: Proceedings 42nd IEEE Symposium on Foundations of Computer Science. pp. 136–145 (2001). https://doi.org/10.1109/SFCS.2001.959888
11. 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)
12. Gennaro, R., Gentry, C., Parno, B., Raykova, M.: Quadratic span programs and succinct nizks without pcps. In: Annual International Conference on the Theory and Applications of Cryptographic Techniques. pp. 626–645. Springer (2013)
13. 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)
14. Groth, J.: On the size of pairing-based non-interactive arguments. In: Annual international conference on the theory and applications of cryptographic techniques. pp. 305–326. Springer (2016)
15. Housni, Y.E., Connor, M., Guillevic, A.: EIP-3026: BW6-761 curve operations [DRAFT], Ethereum Improvement Proposals, no. 3026, [online serial] (October 2020), https://eips.ethereum.org/EIPS/eip-3026
16. Johnson, D., Menezes, A., Vanstone, S.: The elliptic curve digital signature algorithm (ECDSA). Int. J. Inf. Secur. **1**(1), 36–63 (August 2001). https://doi.org/10.1007/s102070100002, https://doi.org/10.1007/s102070100002
17. Kate, A., Zaverucha, G.M., Goldberg, I.: Constant-size commitments to polynomials and their applications. In: International conference on the theory and application of cryptology and information security. pp. 177–194. Springer (2010)
18. Merkle, R.C.: A digital signature based on a conventional encryption function. In: Conference on the theory and application of cryptographic techniques. pp. 369–378. Springer (1987)
19. Reuters: India's 2024 general election sees record participation with 642 million votes cast. https://www.reuters.com/world/india/india-poll-panel-says-642-mln-voters-cast-ballots-general-election-2024-06-03/ (2024), accessed 2026-01-22
20. Sutikno, S., Surya, A., Effendi, R.: An implementation of elgamal elliptic curves cryptosystems. In: IEEE. APCCAS 1998. 1998 IEEE Asia-Pacific Conference on Circuits and Systems. Microelectronics and Integrating Systems. Proceedings (Cat. No.98EX242). pp. 483–486 (1998). https://doi.org/10.1109/APCCAS.1998.743829
21. Vlasov, A., hujw77: EIP-2539: BLS12-377 curve operations [DRAFT], Ethereum Improvement Proposals, no. 2539, [online serial] (February 2020), https://eips.ethereum.org/EIPS/eip-2539
22. WhiteHat, B., Bellés, M., Baylina, J.: ERC-2494: Baby Jubjub elliptic curve [DRAFT], Ethereum Improvement Proposals, no. 2494, [online serial] (January 2020), https://eips.ethereum.org/EIPS/eip-2494
23. Wood, G., et al.: Ethereum: A secure decentralised generalised transaction ledger. Ethereum project yellow paper **151**(2014), 1–32 (2014)

A Cryptographic primitives

In this section, we present the cryptographic primitives used in the protocol described in Section 4. We first introduce elliptic curves as the underlying algebraic setting, and then describe the following building blocks, each tied to a specific role in the system: hash functions and Merkle trees for commitments to structured data; digital signatures for voter authentication; encryption schemes for ballot confidentiality; DKG for decentralizing trust in the election keys; and zero-knowledge proofs (ZK-SNARKs) for verifiability of computations. For each primitive, we also specify the concrete choice of parameters used.

A.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 [8] 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 [23] is chosen because it is the curve supported by Ethereum's precompiled contracts for ZK proof verification, which makes it the optimal choice for verifying SNARK proofs on-chain with minimal gas cost. Then, BabyJubjub [4] is used as an inner curve for elliptic curve operations within arithmetic circuits. Finally, BLS12-377 [7] and BW6-761 [11] 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 ZK-SNARK circuits. This pairing enables succinct verification and composability of ZK-SNARKs within other ZK-SNARKs, 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).
- $r =$ `0x30644e72e131a029b85045b68181585d97816a916871ca8d3c208c16d87cfd47` (254-bit prime).
- $s =$ `0x30644e72e131a029b85045b68181585d2833e84879b9709143e1f593f0000001` (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.

We denote by G^{SEC} the generator of $\mathbb{G}^{\mathsf{SEC}}$ as defined in [8], G^{BN} the generator of $\mathbb{G}_1^{\mathsf{BN}}$ as defined in [23], G^{BJ} the generator of $\mathbb{G}^{\mathsf{BJ}}$ as defined in [22], G^{BW} the generator of $\mathbb{G}_1^{\mathsf{BW}}$ as defined in [15], and G^{BLS} the generator of $\mathbb{G}_1^{\mathsf{BLS}}$ as defined in [21].

- $G^{\mathsf{SEC}} = (\texttt{0x79be667ef9dcbbac55a06295ce870b07029bfcdb2dce28d959f2815b16f81798}, \texttt{0x483ada7726a3c4655da4fbfc0e1108a8fd17b448a68554199c47d08ffb10d4b8})$.
- $G^{\mathsf{BN}} = (\texttt{0x01}, \texttt{0x02})$.
- $G^{\mathsf{BJ}} = (\texttt{0x0b9fefffffffffaaabffffffffffffffefffffffffffffffffffffffffffff}, \texttt{0x17fffffffffffffffffffffffffffffffffffffffffffffffffebaaedce6af48a03bbfd25e8cd0364141})$.
- $G^{\mathsf{BW}} = (\texttt{0x1075b020ea190c8b277ce98a477beaee6a0cfb7551b27f0ee05c54b85f56fc779017ffac15520ac11dbfcd294c2e746a17a54ce47729b905bd71fa0c9ea097103758f9a280ca27f6750dd0356133e82055928aca6af603f4088f3af66e5b43d}, \texttt{0x58b84e0a6fc574e6fd637b45cc2a420f952589884c9ec61a7348d2a2e573a3265909f1af7e0dbac5b8fa1771b5b806cc685d31717a4c55be3fb90b6fc2cdd49f9df141b3053253b2b08119cad0fb93ad1cb2be0b20d2a1bafc8f2db4e95363})$.
- $G^{\mathsf{BLS}} = (\texttt{0x008848defe740a67c8fc6225bf87ff5485951e2caa9d41bb188282c8bd37cb5cd5481512ffcd394eeab9b16eb21be9ef}, \texttt{0x01914a69c5102eff1f674f5d30afeec4bd7fb348ca3e52d96d182ad44fb82305c2fe3d3634a9591afd82de55559c8ea6})$.

A.2 Hash functions

DAVINCI uses different hash functions. On the one side, we use Keccak256 [5] for Ethereum address derivation, and Poseidon [13], MiMC and MiMC-7 [2] for arithmetic circuits within zero-knowledge proofs.

Keccak256. Keccak256 is the standard hash function used by Ethereum and is employed in DAVINCI to compress messages for signing and deriving Ethereum addresses from public keys over $\mathbb{G}^{\mathsf{SEC}}$. Keccak256 takes arbitrary-length bitstrings and outputs 256-bit digests: $\mathsf{Keccak} : \{0,1\}^* \to \{0,1\}^{256}$. This function is not SNARK-friendly and it is used exclusively in the authentication circuit, where the verification of Ethereum-compatible signatures requires reproducing the original message hash computed by Ethereum clients (see Section 4.5.2).

Poseidon. Poseidon is a SNARK-optimized hash function designed for efficient implementation inside arithmetic circuits. It is used in DAVINCI for computing commitments, nullifiers, and the state Merkle tree roots. Poseidon is used as $\mathsf{Poseidon} : \mathfrak{F}_s \to \mathbb{F}_s$, where $\mathfrak{F}_s$ is the set of tuples of $\mathbb{F}_s$-elements of any length, and $\mathbb{F}_s$ is the field described in Section A.1.

A.3 Merkle trees

TODO: Add description of Incremental Merkle trees.

The DAVINCI protocol uses sparse Merkle Trees (SMTs) as its primary data structure for maintaining off-chain state. Following the construction in [3], these are full binary trees of fixed depth in which each leaf encodes a key-value pair and empty leaves are explicitly represented, allowing for both inclusion and

non-inclusion proofs. A key property of SMTs is that the hash of each key determines a unique traversal path from the root to the leaf, regardless of insertion order. When inserting a new key, the tree descends until it encounters either an empty node or a leaf. In case of a path collision, the algorithm splits at the first differing bit, creating intermediate nodes accordingly.

In DAVINCI, SMTs are used to represent two core data structures: the census tree, which encodes the list of eligible voters along with their respective voting weights, and the state tree, which encodes the current status of submitted votes. Both trees have fixed depth 64. The census tree is instantiated with the MiMC hash function over the BLS12-377 scalar field, while the state tree uses Poseidon over the BN254 scalar field. Only the root hash of each tree is published on-chain, ensuring verifiability with minimal on-chain storage. To support state transitions and circuit verification, each tree MT has the following functions associated:

- $\mathsf{MT.Root}()$: it returns the current Merkle root of the tree.
- $\mathsf{MT.Insert}(path, \mathsf{leaf})$: it inserts a new leaf leaf at the position defined by the given path.
- $\mathsf{MT.Update}(\mathsf{leaf_{old}}, \mathsf{leaf})$: it updates an existing leaf in the tree.
- $\mathsf{MT.MembershipProof}(\mathsf{leaf})$: it returns a Merkle proof of inclusion for the given leaf.
- $\mathsf{MT.NonMembershipProof}(\mathsf{leaf})$: it returns a proof that the given leaf is not included in the tree.
- $\mathsf{MT.Verify}(\mathsf{leaf}, \mathsf{Proof}, \mathsf{root})$: it verifies that the given proof corresponds to the claimed root and leaf.

These functions are used by sequencers to construct proofs that certify the validity of state transitions. In particular, they allow for off-chain state evolution that is independently verifiable on-chain and within ZK circuits. Additional details on the structure of the census and state trees, as well as the encoding of their leaves, are provided in Sections 4.3 and 4.4.

A.4 Commitment scheme

Description of KZG [17] with SRS **[add link to Ethereum's BLS12-381 SRS]**, C.Commit(), and C.Verify(). Original methods: C.Setup, C.Open, C.VerifyPoly, C.CreateWitness, C.VerifyEval, C.Prove, C.Verify.

A.5 Digital signature scheme

TODO: add S.ExtractPublicKey that extracts pk from `signature` (in Ethereum it's called `ECRecover`).

DAVINCI uses the elliptic curve digital signature algorithm (ECDSA) [16] 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 Section4.5 for more details). Below, we describe the algorithms, which follow the standard ECDSA protocol. (The output of the hash does not match – check where variables live.)

- $\mathsf{S.Sign_{sk}}(\mathsf{message} \in \{0,1\}^\star)$:
  1. Compute $h = \mathsf{Keccak}(\mathsf{message}) \bmod q$.
  2. Select a random scalar $k \xleftarrow{\$} \mathbb{Z}_q^*$.
  3. Compute $R = k \cdot G^{\mathsf{SEC}} = (x_R, y_R) \in \mathbb{G}^{\mathsf{SEC}}$.
  4. Set $r = x_R \bmod q$.
     - If $r = 0$, go back to step 2.
     - Else, continue.
  5. Compute $s = k^{-1} \cdot (h + r \cdot \mathsf{sk}) \bmod q$.
     - If $s = 0$, go back to step 2.
     - Else, continue.
  6. Output $\sigma = (r, s)$.
- $\mathsf{S.Verify_{pk}}(\mathsf{message} \in \{0,1\}^\star, \sigma = (r,s) \in (\mathbb{Z}_q^*)^2)$:
  1. Check that $\mathsf{pk} \in \mathbb{G}^{\mathsf{SEC}}$.
  2. Compute $h = \mathsf{Keccak}(\mathsf{message}) \bmod q$.
  3. Compute $w = s^{-1} \bmod q$.
  4. Compute $u_1 = h \cdot w \bmod q$.
  5. Compute $u_2 = r \cdot w \bmod q$.
  6. Compute
  $$X = u_1 \cdot G^{\mathsf{SEC}} + u_2 \cdot \mathsf{pk} = (x_X, y_X) \in \mathbb{G}^{\mathsf{SEC}}.$$
  7. Accept if $r = x_X \bmod q$; otherwise, reject.

A.6 Encryption scheme

To preserve ballot secrecy while enabling tallying, DAVINCI employs a threshold variant of the ElGamal cryptosystem instantiated over the BabyJubjub curve [20]. This scheme offers two properties crucial for the protocol: *additive homomorphism*, which allows the aggregation of encrypted votes without decryption, and *re-encryption*, which enables ciphertext randomization without altering the underlying plaintext. Together, these properties allow sequencers to tally votes while preventing voters from producing receipts that could be used for coercion or vote selling. The corresponding public key is generated collectively by the sequencers via the distributed key generation protocol described in Section A.7, ensuring that no single entity can decrypt ballots unilaterally.

Encryption and decryption. Given a message m, the ElGamal encryption algorithm maps it into a group element M of $\mathbb{G}^{\text{BJ}}$ and outputs a ciphertext as follows:

- $\mathsf{Enc}_{\mathsf{pk}}(m; k \in \mathbb{F}_t)$:
  1. Map m to $\mathbb{G}^{\text{BJ}}$ via $M = m \cdot G^{\text{BJ}}$.
  2. Compute $C_1 = k \cdot G^{\text{BJ}} \in \mathbb{G}^{\text{BJ}}$.
  3. Compute $C_2 = k \cdot \mathsf{pk} + M \in \mathbb{G}^{\text{BJ}}$.
  4. Output $\mathsf{ciphertext} = (C_1, C_2) \in (\mathbb{G}^{\text{BJ}})^2$.
- $\mathsf{Dec}_{\mathsf{sk}}(\mathsf{ciphertext})$:
  1. Parse $\mathsf{ciphertext} = (C_1, C_2)$.
  2. Compute $K = \mathsf{sk} \cdot C_1 \in \mathbb{G}^{\text{BJ}}$.
  3. Output $M = C_2 - K$.

Since the plaintext message space is typically small, recovering m from M is efficient: although the mapping $m \mapsto M$ is not generally invertible, it can be reversed by brute-force search or optimized techniques such as baby-step giant-step [6].

Homomorphic addition and reencryption. ElGamal encryption is additively homomorphic. Given two ciphertexts (C_1, C_2) and (C_1', C_2'), their component-wise addition yields another valid ciphertext $(C_1 + C_1',\ C_2 + C_2')$. The resulting ciphertext decrypts to the sum of the two underlying messages. This property allows sequencers to aggregate encrypted ballots directly. Moreover, to prevent linkability and ensure receipt-freeness, sequencers also re-randomize ciphertexts. Re-encryption exploits this property by adding to a ciphertext an encryption of zero. This operation yields a fresh ciphertext of the same message under new randomness, making it computationally infeasible to link the re-encrypted ballot with the original submission.

- $\mathsf{EncAdd}_{\mathsf{pk}}(\mathsf{ciphertext} \in (\mathbb{G}^{\text{BJ}})^2, \mathsf{ciphertext}' \in (\mathbb{G}^{\text{BJ}})^2)$:
  1. Parse $\mathsf{ciphertext} = (C_1, C_2) \in (\mathbb{G}^{\text{BJ}})^2$.
  2. Parse $\mathsf{ciphertext}' = (C_1', C_2') \in (\mathbb{G}^{\text{BJ}})^2$.
  3. Output $(C_1 + C_1', C_2 + C_2') \in (\mathbb{G}^{\text{BJ}})^2$.
- $\mathsf{ReEnc}_{\mathsf{pk}}(\mathsf{ciphertext} \in (\mathbb{G}^{\text{BJ}})^2; k \in \mathbb{F}_t)$:
  1. Parse $\mathsf{ciphertext} = (C_1, C_2) \in (\mathbb{G}^{\text{BJ}})^2$.
  2. Compute $\mathsf{ciphertext}' = \mathsf{Enc}_{\mathsf{pk}}(0; k)$.
  3. Output $\mathsf{EncAdd}_{\mathsf{pk}}(\mathsf{ciphertext}, \mathsf{ciphertext}')$.

In the protocol, re-encryption is applied not only to newly submitted ballots but also to ballots already stored in the state tree. By refreshing the randomness of both new and existing ciphertexts, sequencers ensure that it is indistinguishable whether a ballot has been overwritten or merely re-randomized. This mechanism is essential for guaranteeing receipt-freeness: voters cannot produce a verifiable receipt of their choice, and adversaries cannot detect or prove whether a particular vote has been replaced (see Section 7).

A.7 Distributed key generation scheme

To ensure that no single entity controls the decryption key, DAVINCI employs a DKG protocol to jointly derive the ElGamal encryption key pair used for ballots encryption [1]. Participants who contribute to the DKG ceremony are called *key wardens*, and they are each identified by a unique index i. The outcome is a collective public key (`encryptionKey`) that is published on-chain and used by voters to encrypt their ballots, while the corresponding private key is secret-shared among the key wardens. Only a threshold number t out of n key wardens can later collaborate to decrypt the tally. Unlike classical DKG protocols where shares are exchanged in the clear, our construction uses encrypted shares and ZK-SNARK proofs of correctness. This allows the Ethereum smart contract to verify compliance without learning any of the underlying secrets, ensuring both security and verifiability in a fully decentralized setting.

Let t be the threshold parameter and n the number of key wardens, with $t \leq n$. Let G be a generator of the elliptic curve group of order q. Each participant P_i runs the following procedure.

- GenerateEncryptedShares(i):
  1. Select random scalars $a_{i,0}, \ldots, a_{i,t-1} \xleftarrow{\$} \mathbb{Z}_q$.
  2. Define the polynomial $f_i(x) = \sum_{j=0}^{t-1} a_{i,j} x^j$.
  3. For each $j \in \{0, \ldots, t-1\}$, compute public commitments $C_{i,j} = a_{i,j} \cdot G$.
  4. For each participant P_ℓ with index $\ell \in \{1, \ldots, n\}$, compute the secret share $s_{i,\ell} = f_i(\ell)$.
  5. Encrypt each $s_{i,\ell}$ under P_ℓ's public key using a simplified version of the EC integrated encryption scheme (ECIES), obtaining $EncECIES(s_{i,\ell})$.
  6. Output $\{C_{i,j}\}_{j=0}^{t-1}$ and $\{EncECIES(s_{i,\ell})\}_{\ell=1}^{n}$, together with a ZK-SNARK proof attesting that the encrypted values are consistent with the commitments.

The smart contract verifies the encrypted shares.

- VerifyEncryptedShare($i, \ell, EncECIES(s_{i,\ell}), \{C_{i,j}\}$):
  1. Verify that the ZK-SNARK proof attached to $EncECIES(s_{i,\ell})$ is correct.
  2. If the proof is valid, the share is accepted; otherwise, the participant P_i may be slashed.

After collecting valid encrypted shares, each participant P_j can recover their secret share s_j by decrypting all contributions addressed to them:

- DeriveSecretShare(j):
  1. Decrypt all $EncECIES(s_{i,j})$ values received from other participants.
  2. Compute $s_j = \sum_{i=1}^{n} s_{i,j} \mod q$.
  3. Output s_j.

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

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

This approach ensures that the encryption public key is securely generated in a decentralized way, that each key warden learns only their own secret share, and that the correctness of the entire DKG procedure is verifiable on-chain. Misbehavior, such as submitting invalid shares, can be detected and penalized through the slashing mechanism enforced by the smart contract (see Section 6).

A.8 Zero-knowledge proof systems

Zero-knowledge succinct non-interactive arguments of knowledge (ZK-SNARKs) are a crucial component in ensuring the integrity of the election. Voters generate ZK-SNARK proofs to demonstrate that their encrypted ballots comply with the rules and constraints defined by the election parameters, without revealing any information about their choices. Similarly, sequencers produce proofs to certify the correctness of vote aggregation and state transitions throughout the protocol. P.Prove (), P.Verify (proof, PI) – pkey, vkey.

All ZK circuits in DAVINCI are compiled using the Groth16 proof system [14], a widely adopted ZK-SNARK construction known for its succinctness, efficient verification, and minimal proof size. Although all circuits rely on the same proving system, they are instantiated with different elliptic curves depending on the cryptographic requirements of each phase. Specifically, the vote validity circuit is compiled over the BN254 curve, the census-related circuit uses BLS12-377, and the aggregation circuit is instantiated over BW6-761. Finally, the last circuit used by the sequencer—responsible for generating the final proof of correct tallying and result—is compiled over BN254 as well, since this is the proof that is verified on-chain by the smart contract using Ethereum's native precompiles. Further details on the circuits are provided in Section 4.5 and a summary of the instantiations can be found in Fig. 3.