Condorcet.Vote — Protocol Whitepaper

Cryptographic design, public register, blockchain format, and implementation principles — protocol version 1

The Condorcet.Vote project

Julien Boudry (julien.boudry@condorcet.io)

15 November 2026 — revision 1.2 (draft)

Abstract

Condorcet.Vote is a web platform for organizing preferential (Condorcet-method) and single-choice elections over closed nominative electoral rolls. This whitepaper explains how its voting protocol works and what it guarantees.

Three ideas carry the design:

1. Each ballot is signed in the voter's browser with a linkable ring signature (an election-scoped bLSAG variant over Ristretto255): the ballot is provably legitimate, anonymous within its ring, and double votes are publicly evident — all without voter accounts.
2. Every election publishes its complete register — rings, pseudonymous participation, and all signed ballots — as an append-only, hash-chained, platform-signed sequence of JSON blocks that anyone can mirror, re-verify with stock tools, and re-tally.
3. The platform itself is a declared trusted third party: trusted while votes are being cast, independently auditable afterwards.

The document states precisely what is and is not protected, details the block format and what its immutability does and does not mean, and presents the trade-offs between cryptographic strength and usability as factual engineering decisions.

Contents

1 Introduction **3**
- 1.1 Purpose and scope . . . 3
- 1.2 System overview . . . 3

2 Security objectives and trust model **3**
- 2.1 Guarantees . . . 3
- 2.2 The trusted-third-party model . . . 4
- 2.3 Explicit non-goals . . . 4

3 Cryptographic foundations **4**
- 3.1 Implementation architecture: one Rust core, executed on both sides . . . 4
- 3.2 Primitive inventory . . . 5
- 3.3 The ballot signature: an election-scoped bLSAG variant . . . 6
- 3.4 Proof of ownership (Operation D) . . . 7
- 3.5 Canonical string encodings . . . 8

4 The voting protocol **8**

4.1 Election lifecycle . . . 8
4.2 Key provisioning: three coexisting modes . . . 9
4.3 Rings . . . 10
4.4 Key distribution (server-generated mode) . . . 12
4.5 The ballot: a byte-exact signed payload . . . 12
4.6 Casting and double-vote prevention . . . 13
4.7 Participation receipt and individual verifiability . . . 13
4.8 Election options with explicit privacy consequences . . . 14

5 The public register and its blockchain **14**
5.1 Two deliberately unlinkable sides . . . 14
5.2 Ballot storage: byte-exact and append-only . . . 15
5.3 Anti-chronology rules . . . 15
5.4 The chain: envelope and linkage . . . 15
5.5 Block types . . . 16
5.6 Serialization: sign the file, not a canonical model . . . 19
5.7 Sealing and anchoring: how a block is produced . . . 19
5.8 Platform signature and key management . . . 20
5.9 External time anchor: OpenTimestamps . . . 20
5.10 What "immutable" means here — and what it does not . . . 21
5.11 What anyone can verify . . . 21
5.12 Tally reproducibility . . . 22

6 Threat model **22**
6.1 Defended threats . . . 22
6.2 Accepted limitations — documented, not hidden . . . 23

7 Implementation safeguards **24**
7.1 Protecting the signing window . . . 24
7.2 Data minimization and chronology hygiene . . . 24
7.3 The database as the single arbiter . . . 24
7.4 Publication isolation . . . 25
7.5 Engineering discipline . . . 25

8 Design trade-offs: advanced cryptography versus usability **25**
8.1 Verifiability by publication, not by advanced tally cryptography . . . 25
8.2 Receipts over receipt-freeness . . . 25
8.3 Server-generated keys by default, BYO-only as the trust-minimizing mode . . . 25
8.4 Account-free voting through emailed capability links . . . 26
8.5 A capped ring size: one ceiling, one launch size per election . . . 26
8.6 The distribution re-delivery cache . . . 26
8.7 A central platform with mirrorable evidence, not a decentralized ledger . . . 26

9 Known limitations and planned work **27**

10 References **27**

Status of this document. This whitepaper is a *secondary source*: a consolidated summary of the project's normative specifications and of the implementation (the platform source code and the `cryptoVote` library). It is descriptive, not normative. If anything in this document disagrees with the specifications or the code, the specifications and the code prevail. It describes **protocol version 1**.

1 Introduction

1.1 Purpose and scope

This document explains the **voting protocol** of Condorcet.Vote and the implementation principles that uphold it: the cryptographic mechanisms (precisely), how ballots are stored, how the public register is formatted, chained, signed, and verified, which guarantees follow, and which threats are explicitly *not* covered. It is written for technically inclined organizers, security reviewers, and independent auditors.

It is a protocol and implementation document, not a general security audit of the platform: organizer account management, web application hardening, and infrastructure operations are out of its scope. Only the implementation safeguards that directly carry a protocol guarantee — such as the integrity of the client-side signing environment or the erasure of storage-level chronology — are described, and at the level of principles.

1.2 System overview

Condorcet.Vote runs elections for closed, nominative electoral rolls: an organizer defines candidates and voters; voters receive a personal access link (*VoteLink*) by email and vote from their browser without creating an account. Ballots are ranked expressions tallied by Condorcet methods (or single-choice, first-past-the-post), computed by the open-source CondorcetPHP engine.

The protocol rests on three pillars:

1. **Ballot anonymity through linkable ring signatures.** Each ballot is signed in the voter's browser with a one-time secret key whose public key is hidden among hundreds of others in a *ring*. The signature proves the ballot comes from a legitimate ring member without revealing which one.
2. **Public verifiability through publication.** Every election publishes a *public register* — its frozen definition, all rings, a pseudonymous participation register, and every ballot with its signature — distributed as an append-only, hash-chained, platform-signed *blockchain* of JSON artifacts that anyone can download, mirror, and re-verify with standard tools. Anyone can recompute the tally.
3. **A declared trusted-third-party model.** The platform is trusted at vote time (it necessarily sees a ballot arrive on an authenticated session) and is *not* trusted afterwards: everything it did is checkable from the published register alone.

2 Security objectives and trust model

2.1 Guarantees

The protocol provides, simultaneously and without voter accounts:

- **Ballot confidentiality.** No member of the public, no organizer, and no post-hoc reader of a logical database export can link a stored ballot to a voter. The cryptographic anonymity set of a ballot is its ring; the effective anonymity set is the ring's *participating* members (see §4.3 and §6).
- **Public verifiability.** Anyone can verify, from the published register alone: that every ballot carries a valid ring signature, that no key signed twice, that ballots do not out-

number recorded voters, and that the published result is the deterministic tally of the published ballots (see §5.11).
- **Double-vote resistance.** One effective ballot per voter, enforced transactionally by the participation register and evidenced publicly by cryptographic *key images* (see §4.6).

2.2 The trusted-third-party model

The platform is a **trusted third party with bounded trust in time**:

- **Trusted at vote time.** The platform serves the voting page, delivers secret keys in its default provisioning mode, and receives each signed ballot over the voter's authenticated VoteLink session. During this window it transiently knows who is voting and, in the default mode, which key it handed to whom. This exposure is a deliberate, documented design decision, not an oversight (§8).
- **Untrusted afterwards.** Every claim the platform makes is materialized in the public register and independently checkable. The register's two sides — participation and ballots — are published so that they cannot be joined into a voter-to-ballot link.

The amount of trust granted at vote time is itself an **election-level dial**: an election can be configured as *BYO-only* (§4.2), in which case the platform never holds any voter's signing secret and the trusted window narrows to ballot reception alone.

2.3 Explicit non-goals

The following are consciously **out of scope** for protocol version 1 and are treated at length in §6 and §8:

- **Coercion resistance / receipt-freeness.** Voters obtain provable receipts; published ballots are cleartext rankings. The protocol targets private and associative elections, not coercion-sensitive public ones.
- **Decentralization.** The chain is append-only and hash-linked but produced by a single operator; immutability rests on independent mirroring (§5.10), complemented by a per-block Bitcoin time anchor via OpenTimestamps (§5.9) — proof of existence, not distributed consensus.
- **Trustless operation.** The design does not attempt to remove the platform from the trust equation at vote time (no mix-nets, no homomorphic or zero-knowledge tally).

3 Cryptographic foundations

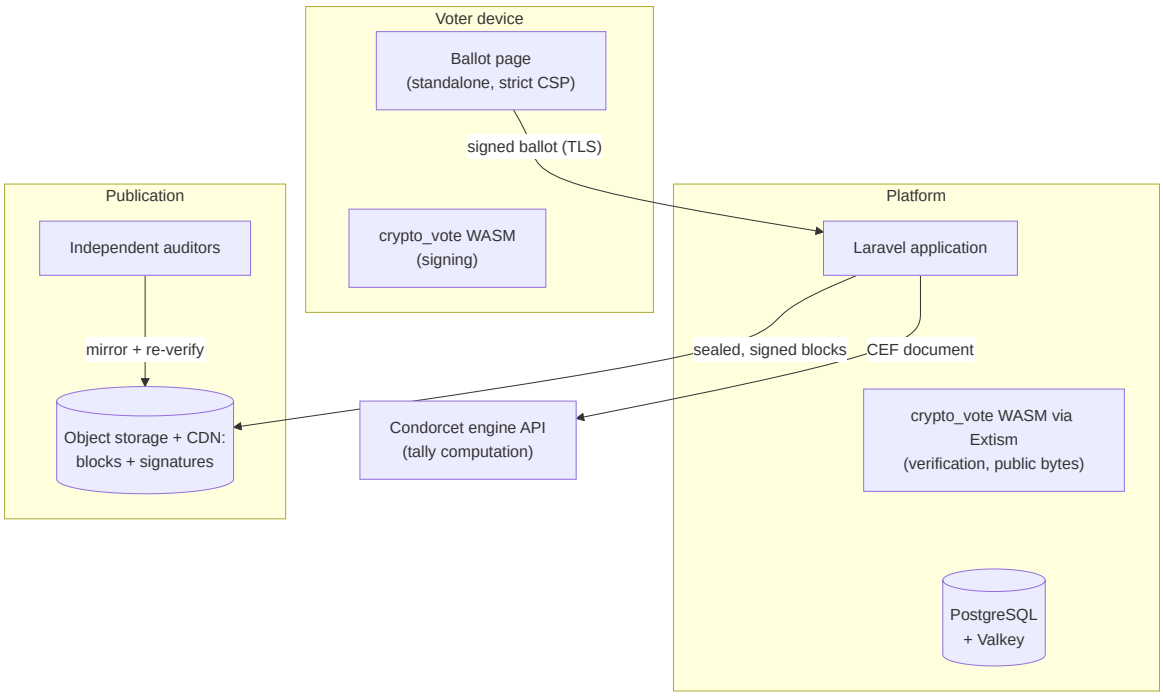
3.1 Implementation architecture: one Rust core, executed on both sides

All ballot cryptography is implemented once, in a dedicated open-source Rust crate — `crypto_vote` (repository *cryptoVote*, AGPL-3.0-or-later) — and compiled to WebAssembly for both execution environments. The same byte-for-byte logic therefore runs in the browser and on the server; the platform never re-implements a primitive in PHP or JavaScript.

- **Voter device — signing.** The npm package `@condorcet.vote/crypto-vote` (a `wasm-bindgen` ES module) runs in the browser, on a standalone voting page served under the platform's strictest CSP.
- **Platform — verification.** The Composer package `condorcet-vote/crypto-vote-php` wraps an Extism build (`wasm32-wasip1`) of the same core and runs it in-process on the

server; it handles only public bytes. There is no separate crypto micro-service.

The crate exposes a deliberately minimal oracle of four operations: `generate_identity`, `sign_vote`, `verify_vote`, and the voter-initiated ownership proof (§3.4). It knows nothing about elections, databases, or networks; double-vote bookkeeping belongs to the platform.



3.2 Primitive inventory

Purpose	Primitive	Parameters
Group	Ristretto255 (prime-order group over Curve25519)	prime order $\ell \approx 2^{252}$; `curve25519-dalek`
Ballot signature	Linkable ring signature — election-scoped bLSAG variant	Fiat–Shamir, $32(1+n)$-byte signatures for ring size n
Hash to scalar / to point	BLAKE2b-512	512-bit wide reduction mod ℓ; Ristretto one-way map
Key image (linkability tag)	$I = x \cdot H_p(\text{domain} \parallel \text{len}(e) \parallel e \parallel P)$	election-scoped; 32 bytes
Ownership proof (Operation D)	Chaum–Pedersen DLEQ, non-interactive	64-byte proof; fresh 32-byte nonce
Artifact string checksum	BLAKE3, truncated to 4 bytes	typo/mix-up guard only — never a security primitive
Chain block hash	SHA3-512 over the stored block bytes	lowercase hex, 128 chars

Purpose	Primitive	Parameters
Participation email digest (opt-in)	SHA3-256(election id ‖ normalized email)	deliberately weak pseudonymization, see §4.8
Chain block signature	OpenPGP v6 (RFC 9580), Ed25519, detached	produced by the Sequoia `sq` CLI
VoteLink token at rest	SHA3-256 digest	plaintext token exists only in the emailed URL
Stored secrets at rest	AES-256-GCM (application-key encryption)	voting secret keys, pre-distribution
Randomness	OS CSPRNG	`getrandom(2)` family; Web Crypto `getRandomValues` in browsers; WASI `random_get`

Notation used below: G is the Ristretto255 base point; ℓ the group order; H_s hashing to a scalar (BLAKE2b-512 then wide reduction); H_p hashing to a group element (BLAKE2b-512 then the Ristretto one-way map); enc$(\cdot)$ the 32-byte canonical compressed encoding of a point; $\text{len}_8(\cdot)$ a 64-bit big-endian byte-length prefix; $\|$ byte concatenation.

3.3 The ballot signature: an election-scoped bLSAG variant

The scheme is a **linkable spontaneous anonymous group (LSAG) signature** in the *back-linkable* form (bLSAG) documented in the Monero literature, whose ring equations follow Liu–Wei–Wong [LWW04] as presented in *Zero to Monero* [ZtM20]. It is Schnorr-based: the signer's link in the ring uses the Schnorr response $s_\pi = \alpha - c_\pi x$, chained around all ring members so the signature verifies for the whole ring without revealing which member signed.

Key generation. A voting identity is a uniformly random secret scalar $x \in [1, \ell)$ with public key $P = xG$. There is no clamping and no key derivation schedule; the secret is generated from the OS CSPRNG (in the voter's browser for self-enrolled keys, on the platform for server-generated keys).

Election-scoped linkability base. For a public key P and an election identifier e (the election's UUID, NFC-normalized), the scheme defines a per-member base point:

$$B_P \;=\; H_p(\texttt{"crypto_vote:blsag:key-image-point"} \,\|\, \text{len}_8(e) \,\|\, e \,\|\, \text{enc}(P))$$

The signer's **key image** is $I = x \cdot B_P$. Standard bLSAG uses $I = x \cdot H_p(P)$; folding the election identifier into the hash-to-point pre-image is this scheme's *only* deviation from the standard construction. Its purpose: the same secret key produces the *same* key image within one election (same-election linkability — double votes with the same key are publicly evident) but *unlinkable* key images across different elections (under the decisional Diffie–Hellman assumption in the Ristretto255 group, key images of one key in two elections cannot be correlated without x).

Message binding. The signed message is not the raw ballot but the election-bound blob

$$m \;=\; \text{len}_8(e) \,\|\, e \,\|\, v$$

where v is the ballot payload bytes (§4.5), treated as opaque and never normalized. The length prefix removes concatenation ambiguity between e and v. Only e is NFC-normalized (a no-op for the ASCII UUIDs used in practice).

Ring canonicalization. A ring is the set of $n \geq 2$ authorized public keys, including the signer's. Signer and verifier independently sort the ring lexicographically by 32-byte compressed encoding, so wire order is irrelevant; duplicate members are rejected, as are non-canonical point encodings and the identity element.

Signing (signer at index π after canonicalization):

1. Sample $\alpha \xleftarrow{\$} \mathbb{Z}_\ell$ and decoy responses $s_i \xleftarrow{\$} \mathbb{Z}_\ell$ for all $i \neq \pi$.
2. Seed the challenge chain: $c_{\pi+1} = H_s(D_c \parallel \text{len}_8(m) \parallel m \parallel \text{enc}(\alpha G) \parallel \text{enc}(\alpha B_{P_\pi}))$ with domain string $D_c = \texttt{"crypto_vote:blsag:challenge"}$.
3. For each subsequent member $i = \pi+1, \pi+2, \ldots$ (indices mod n), compute $L_i = s_i G + c_i P_i$, $R_i = s_i B_{P_i} + c_i I$, and $c_{i+1} = H_s(D_c \parallel \text{len}_8(m) \parallel m \parallel \text{enc}(L_i) \parallel \text{enc}(R_i))$, stopping once c_π has been produced.
4. Close the ring: $s_\pi = \alpha - c_\pi x \bmod \ell$.

The signature is $\sigma = (c_0, s_0, \ldots, s_{n-1})$ together with the key image I — $32(1+n)$ bytes plus 32 bytes of key image.

Verification. Given $(\sigma, I, \text{ring}, e, m)$: recompute the challenge chain from c_0 across all n members using the same equations and accept iff it closes back onto c_0. Note that each verification step recomputes the per-member base B_{P_i}, so the election identifier is bound into every link of the chain twice: through the message m and through the linkability bases. Verification is a total function: malformed inputs (wrong lengths, non-canonical encodings, ring mismatch) return *false* rather than raising errors.

Security properties:

- *Anonymity:* the verifier learns only that *some* ring member signed; the distribution of (c, s_i) values is identical for every possible signer.
- *Linkability:* two signatures by the same secret key in the same election necessarily carry the same key image I.
- *Non-slanderability / unforgeability:* producing a valid signature or a colliding key image without a ring member's secret reduces to the discrete logarithm problem in Ristretto255 (standard LSAG arguments).
- *Cross-election unlinkability:* added by the variant, argued under DDH as above.

Implementation notes. Scalar and point arithmetic is constant-time (`curve25519-dalek`); ring sorting and index lookup operate on public data only. Secret scalars are zeroized on drop; the browser boundary passes the secret as a mutable `Uint8Array` that the caller wipes. The crate ships frozen test vectors that pin the domain strings, the binding format, and the signature/key-image bytes against silent protocol drift, plus bit-flip malleability sweeps, fuzz targets over all parsers, and a CI `cargo audit`.

3.4 Proof of ownership (Operation D)

An optional, **voter-initiated** operation lets the holder of a secret key prove to any third party that a given key image — hence a given published ballot — is theirs, *without revealing the secret key.* It is a non-interactive **Chaum–Pedersen proof of discrete-logarithm equality**

[CP92]: knowledge of one scalar x such that $P = xG$ and $I = xB$, where B is the same election-scoped base as above.

- The verifier supplies a fresh 32-byte nonce η (freshness, not designated verification).
- Prover: $r \xleftarrow{\$} \mathbb{Z}_\ell$; $R_1 = rG$; $R_2 = rB$; $c = H_s(D_o \parallel \text{len}_8(e) \parallel e \parallel \text{len}_8(\eta) \parallel \eta \parallel \text{enc}(P) \parallel \text{enc}(B) \parallel \text{enc}(I) \parallel \text{enc}(R_1) \parallel \text{enc}(R_2))$; $s = r + cx$. The proof is (c, s) — 64 bytes. $D_o =$ `"crypto_vote:ownership:challenge"` is distinct from the ring-signature domain, preventing cross-protocol replay.
- Verifier: recompute $R'_1 = sG - cP$, $R'_2 = sB - cI$, rederive the challenge and compare.

Two properties are stated bluntly because they matter to voters: the proof is **transferable** (anyone holding the nonce and proof can convince anyone else — it is not a designated-verifier proof), and using it **reveals how the prover voted** (it links the prover to a published cleartext ballot). It adds no coercion exposure that the receipt (§4.7) did not already create; it standardizes an existing capability for legitimate use cases such as mandated or proxy voting. It is served by a public tools page, runs entirely client-side, and is outside any organizer control.

3.5 Canonical string encodings

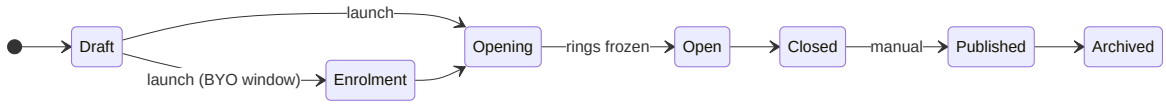
Every cryptographic artifact crossing a UI, API, or file boundary uses a prefixed, checksummed canonical form:

`<tag>_<lowercase hex body>_<8 hex chars of checksum>`

with tags `pk` (public key), `sk` (secret key), `ki` (key image), `blsag` (ring signature), `own` (ownership proof), `nonce` (proof nonce). The checksum is the first 4 bytes of `BLAKE3("crypto_vote/prefixed-checksum/v1" || tag || 0x00 || payload)`, so a mistyped value — or a `pk_` pasted where a `ki_` was expected — is rejected immediately. The checksum is an integrity and type-confusion guard only; authenticity always comes from the bLSAG proof. The WASM, Extism, and CLI front ends accept this form exclusively and reject bare hex.

4 The voting protocol

4.1 Election lifecycle



An election may be *cancelled* from any pre-terminal state; cancellation withdraws the public views but cannot retract already-mirrored chain blocks (§5.5). The `Opening` state is a transient, all-or-nothing provisioning build: server-side keys are minted, pooled with imported and enrolled public keys, shuffled, partitioned into rings, and frozen. The transition to `Open` is gated on every ring being confirmed frozen — a failed build never yields a half-open election. At `Closed`, every remaining stored secret is purged. While `Open`, the roll of a standalone election that allows server-generated keys may still **grow**: voters can be added — never edited or removed — each addition being definitive, server-provisioned by construction (a frozen ring cannot take a brought key, which is also why a BYO-only election refuses additions), delivered its link at once, and absorbed by the ring headroom and emergency rings of §4.3.

4.2 Key provisioning: three coexisting modes

Mode	Secret key held by	What it adds
Server-generated (default)	Platform (encrypted at rest), then the voter after an explicit distribution	Zero-friction voting; requires trusting the platform at distribution time
Organizer-imported	Voter (the platform never sees the secret)	The platform never knew the secret and can never sign for this voter. The organizer supplied the public key and may — or may not — also hold the secret; the protocol cannot prove either way
Voter-enrolled (bring-your-own)	Voter (generated in their own browser)	Strongest: the platform never knew the secret and can never sign for this voter, and the secret never existed outside the voter's device — the organizer never held it either

All three modes coexist within one election — a **default election is mixed by construction**: each voter is, individually, server-provisioned, organizer-imported, or self-enrolled, and the server-generated mode is simply the fallback for voters who bring no key. A voter's mode is frozen at the `Opening` build, with no cross-mode fallback (a lost self-managed key means no vote, exactly like a no-show — the unused public key remains in the ring as harmless padding). Public keys are unique per electoral roll, and staged keys are encrypted at rest with a clear SHA3-256 digest as the uniqueness anchor. The mode is not private bookkeeping: it is **published per voter** in the final participation register (§5.5), so which voters the platform ever held a key for is itself a public, auditable fact.

4.2.1 BYO-only elections: removing the platform from key custody

Mandatory enrolment (*BYO-only*) is a **first-class election mode**, not a side option: a single setting made at draft time turns the enrolment window mandatory, and every voter must then bring an imported key or enrol one — the browser generates the key pair locally (WASM) and hands the voter a key file to keep. **No server-side key is ever generated**: the platform never holds any voter's signing secret.

The mode is trivial to enable for the organizer and more demanding for the voters — each is responsible for keeping a key, and a lost key means no vote, exactly like a no-show. In exchange, it *structurally eliminates* several of the trade-offs the default mode accepts:

Property	Default election (mixed; server keys allowed)	BYO-only election
Platform holds voting secrets	For server-provisioned voters — encrypted, erased at distribution	Never
Platform could sign in a voter's stead	Possible for server-generated keys only	Structurally impossible
Key-image precomputation by a malicious platform (§6.2)	Possible for server-generated keys only	Structurally impossible
Distribution machinery — 3-key cap, claims (§4.4)	Required	Absent: nothing to distribute
Transient voter-to-key binding in cache (§8.6)	Exists for server-provisioned voters, up to 180 minutes	Never exists
Ring composition	One launch size per election (100–600 keys) padded with server keys; emergency rings at the 600 ceiling; probabilistic assignment	Known membership; deterministic, even partition; ceiling only
Sparse-ring risk (§6.2)	Mitigated probabilistically (emergency rings, anonymity floor)	Structurally minimized by the balanced partition
Voter burden	Optional — a key is delivered on demand unless the voter brings one	Voter generates, stores, and presents their own key; loss means no vote
Voter-to-ballot exposure at submission	Platform transiently sees it	Unchanged — inherent to authenticated submission

What BYO-only does **not** change is equally explicit: the platform still sees each ballot arrive on an authenticated session (last table row), receipts remain provable, and the publication model is identical — the public-side guarantees (rings, election-scoped key images, register bounds) are the same in both modes.

4.3 Rings

- **One ceiling per election, materialized.** The ring-size ceiling (`max_ring_size`, currently 600) is copied onto the election at the `Opening` build; no ring of that election ever exceeds it, and every later ring reads the materialized value, so a later change of the code constant never affects an in-flight election. At $n = 600$ a ring's public key list is about 19 KB, a signature about 19 KB, and signing or verifying takes roughly 0.5–0.6 s of single-core time — a deliberate cost envelope (§8.5); at the 100-key floor every figure is about six times smaller.
- **Launch provisioning.** For an electoral roll of N voters, the election opens with $R_0 = \max(1, \lceil 1.5 \cdot N / \text{max_ring_size} \rceil)$ rings: 1.5 keys per voter models expected demand (most

voters claim one key; turnout is below 100%), not the worst case of the 3-distribution cap.

- **One launch size per election, under the ceiling.** All launch rings share a single size derived from the frozen roll — N_s voters without an imported or enrolled key, N_b imported or enrolled keys:

$$\text{launch_size} \;=\; \text{clamp}\Big(\Big\lceil \tfrac{3\cdot 2\cdot N_s + N_b}{R_0} \Big\rceil,\; 100,\; \text{max_ring_size}\Big)$$

  $3 \cdot N_s$ is the worst-case redistribution demand (every server-provisioned voter burning all three attempts), the factor 2 leaves room for voters added mid-election (a launch-only allowance: mid-election provisioning sizes on the demand it sees and never anticipates a further wave), and each known key takes one slot. Fifty server voters and five enrolled keys open on one ring of 305 keys; three voters open on the 100-key floor; from 100 server voters upward the ring is the full 600. Whenever the target fits under the ceiling, capacity covers the worst case: with a frozen roll such an election can never need an emergency ring, whatever its voters burn.

- **The floor is headroom, not anonymity.** Padding keys never add anonymity (§4.3, last bullet): a three-voter election has an anonymity set of three whatever its ring holds. The 100-key floor keeps room in a tiny launch ring for voters added later to *dilute* its first participants — the deficit-weighted assignment keeps sending part of the new distributions to it — instead of overflowing straight into an emergency ring.

- **Assignment.** At distribution time a ring is picked among active rings with free keys, with probability proportional to each ring's distribution deficit, floored by a uniform component. With k active rings, deficits $I_i = D_{max} - D_i$ and $S = \sum_j I_j$:

$$P_i \;=\; 0.25 \cdot \tfrac{1}{k} \;+\; 0.75 \cdot \tfrac{I_i}{S} \qquad (S > 0;\ \text{uniform if } S = 0)$$

  A freshly created ring therefore captures most of the next distributions and crosses its useful anonymity threshold organically, then decays back to uniform.

- **Emergency rings: always the ceiling, as many as needed at once.** Whenever free keys would drop below projected remaining demand (unvoted server-provisioned voters times a 1.3 factor) — deliberately early, because a sparse ring is costly and an extra ring is cheap — a queued job computes how many full `max_ring_size` rings the shortfall requires, capped by the hard ceiling of 3 keys per voter over the *current* roll, and creates them all in one transaction. An emergency is an abnormal event (a roll growing mid-election), so it takes the maximum headroom: a three-voter election that suddenly receives 2 000 voters gets five 600-key rings in a single pass, not a trickle of tiny rings, each of which would be a `rings` block, a "later-than" signal and a small anonymity set. Distribution never produces a partial or undersized ring.

- **Anonymity floor.** A ring's effective anonymity set is its *participating* members, not its key count. The design targets a minimum of 25 participants per ring; the target is a design rationale (it shaped the assignment formula and the early emergency trigger), not a runtime parameter, and closing an election is never blocked by it. Padding keys add headroom against ring-draining attacks; they add no anonymity.

4.4 Key distribution (server-generated mode)

- Pre-generated secrets are stored encrypted (AES-256-GCM under the application key) and are **deleted at the moment of distribution**: the claim of a random free key (a `FOR UPDATE SKIP LOCKED` pick) and the nulling of its stored ciphertext happen in one statement, inside one short transaction that also enforces the per-voter cap with a guarded, race-proof write. No durable voter-to-key correspondence is ever stored.
- Distribution requires an explicit voter action, is capped at **3 keys per voter** by default (organizer-granted extras are possible and publicly visible in the register), and the secret is delivered once over TLS.
- For re-delivery convenience, the distributed secret lives for at most **180 minutes** in a voter-keyed, encrypted cache entry (Valkey), returned unchanged if the voter reloads the voting page before voting; the entry is deleted immediately when a ballot is accepted. For its lifetime this cache *is* a transient voter-to-key binding — an explicitly accepted usability trade-off within the trusted-at-vote-time model (§8.6).
- Closing the election enqueues a purge of every remaining stored secret; the "no key issued after close" guarantee rests on the state gate, not on the purge's timing.

4.5 The ballot: a byte-exact signed payload

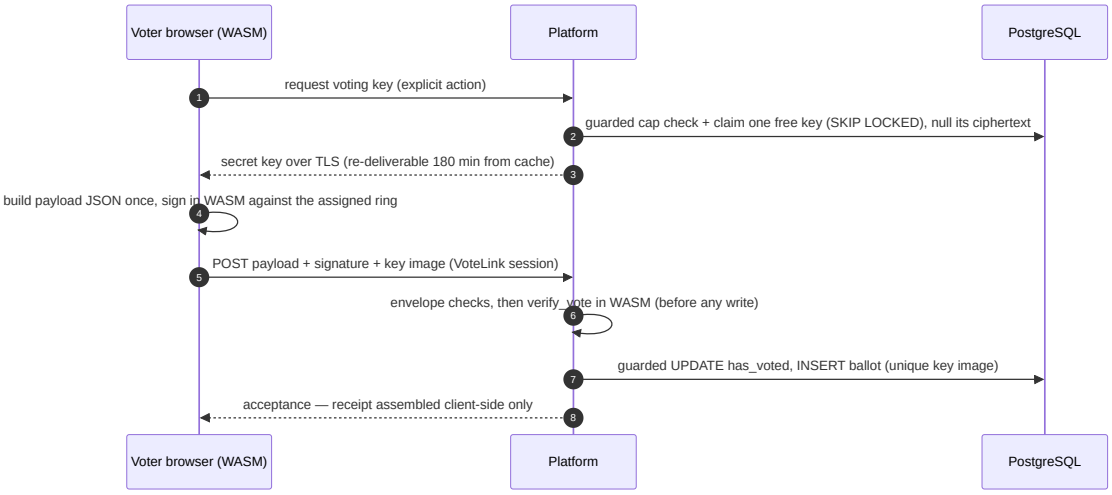
The ballot is a JSON document **built exactly once in the voter's browser**, signed byte-for-byte, transmitted unchanged, and stored verbatim (§5.2). The stored bytes are the canonical ballot — "what you sign is what you store is what you verify".

```
{
  "v": 1,
  "election": "0c2e7e2a-5df3-4c0e-9d2b-3f6a8f0c9e21",
  "ballot": "Annecy > Bourget = Leman",
  "weight": 1
}
```

- `v` — the election's protocol version (currently 1).
- `election` — the election UUID; also the `election_id` signing parameter.
- `ballot` — the voter's expression as a flat UTF-8 string using CEF ranking syntax (candidate labels with `>` and `=`), or the `/EMPTY_RANKING/` sentinel for an explicit blank.
- `weight` — always present, integer ≥ 1; the voter's authoritative ballot weight, validated against the electoral roll at submission.
- `voter` — present only in non-anonymous elections (§4.8): the voter's public voter ID.

The server validates the **envelope** (JSON validity and flatness, an 8192-byte cap, version and election match, weight authorization, voter match in non-anonymous mode) but deliberately never validates the `ballot` content: a malformed expression deterministically classifies as a null ballot at tally rather than being rejected — the classification rule is published and reproducible (§5.12). No JSON canonicalization is imposed; the signature covers exact bytes, and the serialization path into the tally rejects every reserved character that could smuggle weights or extra votes into the CEF document.

4.6 Casting and double-vote prevention



Server-side checks run in this order: election is `Open`; the voter's participation record still shows `has_voted = false`; the envelope is valid; the key image is unseen (a database uniqueness constraint on (election, key image) is the race-proof backstop); the ring signature verifies against the ring named by the submission. Acceptance then flips `has_voted` with a guarded conditional update and inserts the ballot row — random UUID, verbatim payload bytes, signature, key image, ring id, **no timestamps** — in one transaction.

Two mechanisms deliberately share the double-vote burden:

- The **participation register is the primary guard**: one voter may hold several valid keys (multiple distributions, or a distribution plus an enrolled key), and key images cannot link ballots signed with *different* keys. Only the voter-scoped `has_voted` flag closes that case, which is why ballot submission is coupled to the VoteLink session and cannot be an anonymous drop-box API.
- The **key image is the public, mathematical evidence**: within the register, key-image uniqueness proves no *key* was counted twice — a check any third party can run without trusting the platform (§5.11).

A submitted ballot is final: no re-vote, no edit.

4.7 Participation receipt and individual verifiability

Once the ballot is accepted, the browser — and only the browser — assembles a receipt containing the election UUID, ring id, voter ID, the voter's key pair, and the ballot's signature and key image. It is displayed once with an optional file download, never emailed, never stored server-side, and impossible to re-issue (the platform no longer holds the secret). The downloaded file is a single UTF-8 JSON object (`.condorcet_receipt`), every cryptographic value in prefixed canonical form and the signed payload carried **verbatim**:

```
{
  "version": "condorcet-receipt/1",
  "election": "0c2e7e2a-5df3-4c0e-9d2b-3f6a8f0c9e21",
  "ring": "<ring id>",
  "voter_id": "<the voter's voter ID>",
  "public_key": "pk_...",
```

```
  "secret_key": "sk_...",
  "ballot": {
    "signed_payload": "<the exact JSON string that was signed, byte-for-byte>",
    "signature": "blsag_...",
    "key_image": "ki_..."
  }
}
```

With the secret key the voter can recompute their key image, locate *their* ballot in the published register, and compare its bytes — individual verifiability, including offline re-verification of the signature. The voting page keeps the secret in memory only until navigation and offers a permanent "hide details" action for shared devices.

The deliberate consequence: receipts are *provable* (see Operation D), so the protocol has **no coercion resistance** — an accepted decision discussed in §8.2.

4.8 Election options with explicit privacy consequences

Three organizer options, all off by default and frozen at launch, trade privacy for institutional requirements. Each triggers an explicit voter notice on the ballot page:

- **Nominative participation disclosure** publishes, for voters who actually voted, their clear name and an election-scoped email digest (SHA3-256 of the election id and normalized address). Abstainers are never disclosed. The digest is deliberately weak pseudonymization — anyone who already holds a candidate email can confirm participation; its purpose is electoral-roll auditing, not concealment.
- **Non-anonymous voting** embeds the voter ID inside the signed payload, publicly linking each ballot to a voter ID; it forces the nominative disclosure on, completing a public ballot-to-name chain. Intended for votes that are attributable by design (e.g. co-ownership assemblies). The embedded voter ID is server-validated, not cryptographically proven — a ring signature hides the signer, so the register alone cannot prove the claim; this is documented rather than hidden.
- **Weighted voting** attaches an integer weight to every voter, published on both register sides and bound inside each signed ballot; the weighted tally is verifiable by multiset equality without linking ballots to voters. In an *anonymous* weighted election a near-unique weight value re-identifies its ballot (k-anonymity among equal weights) — the organizer is warned accordingly.

5 The public register and its blockchain

5.1 Two deliberately unlinkable sides

Every election publishes a **public register** consisting of: (1) its frozen definition; (2) every ring's complete public-key list; (3) a **participation register** — one line per voter, identified only by a random UUIDv4 voter ID, carrying its key-provisioning mode (`server_generated`, `organizer_imported`, or `voter_enrolled` — the distinction is public, per voter), distribution count, voted flag, voted ring, and weight; and (4) a **ballot register** — one record per ballot: ring id, verbatim signed payload, signature, key image. Nothing published links a ballot to a voter ID (outside the opt-in modes of §4.8), and the two sides use different canonical orderings.

5.2 Ballot storage: byte-exact and append-only

The stored ballot bytes are the single source of truth for verification, audit, and tally; the storage rules exist to keep them bit-identical from signature to publication:

- The payload is stored **verbatim in a binary column** (`bytea`), never in a normalizing JSON type: PostgreSQL's `jsonb` reorders keys and rewrites whitespace, which would break byte-exact signature verification. Derived, parsed views may be cached separately; the bytes remain canonical.
- Ballot rows carry **random UUIDv4 identifiers and no timestamps** — there is deliberately no ordered id and no creation datetime to leak casting order.
- The ballots and chain-anchor tables are **append-only by database trigger**: any `UPDATE` or `DELETE` raises an exception. The two codified exceptions are the ballot's `is_chain_published` flag, flipped by the block builder in the same transaction that commits a block — it is also the *only* gate both the human register page and the blockchain read, so the two surfaces always expose the identical ballot set — and the chain anchor's OpenTimestamps status (§5.9), one-way `pending` → `completed`.
- Uniqueness constraints — (election, key image) on ballots and (election, public key) on ring members — are database-level backstops that hold even against application bugs.

5.3 Anti-chronology rules

Vote timing is treated as an anonymity-relevant side channel and suppressed end to end:

- Ballots are published **only in batches of at least 600 records** (the publication batch floor — one average French polling station), on a randomized schedule, sorted by key image inside each block; the remainder flushes only at close.
- The participation register is published **once**, as a single snapshot in the final block, so no participation line shares a publication window with any ballot.
- No published record carries a timestamp; sorting inside every block is deterministic and content-derived (byte order), never arrival order.
- The chain still reveals a coarse partial order (a ballot in block k was cast before block $k{+}1$ was sealed) at the ≥ 600 granularity, and an emergency ring's first appearance is a coarse "later-than" signal for its members. Both residuals are documented and accepted.

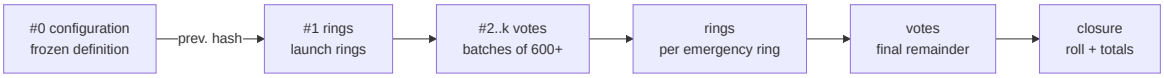
5.4 The chain: envelope and linkage

The register is distributed as an **append-only, hash-chained sequence of platform-signed JSON files** ("blockchain-like" means exactly that: append-only plus hash-chained — *not* decentralized, *not* externally anchored). Every block, regardless of type, opens with the same envelope:

```
{
  "protocol_version": 1,
  "type": "configuration | rings | votes | closure",
  "election_id": "0c2e7e2a-5df3-4c0e-9d2b-3f6a8f0c9e21",
  "block_number": 0,
  "previous_hash": "null for block 0, else 128 hex chars (SHA3-512)"
}
```

- `previous_hash` of block k is the **SHA3-512 of block $k{-}1$'s stored bytes** (lowercase hex); it is `null` only for the genesis block.

- `block_number` is a consecutive per-election sequence starting at 0.
- Blocks are only ever appended at the head; nothing is inserted before an existing block and no hash is ever rewritten. An emergency ring created mid-election is simply a `rings` block appended between two `votes` blocks.
- The block's own hash and its signature are **not** inside the JSON (a document cannot contain its own digest); they live in the storage anchor and in a detached sidecar file (§5.7, §5.8).



5.5 Block types

Four block types exist, in a fixed lifecycle; the authoritative shapes are JSON Schemas (Draft 2020-12), versioned per protocol version and shared by the writing and reading sides. They are published, together with the field-by-field format reference and the verification procedure, on the platform's *Blockchain protocol & formats* page (https://condorcet.vote/blockchain-protocol). The election **result is never a block**: it is a deterministic recomputation from the published blocks.

`configuration` — block 0 (genesis). The frozen election definition: title, organizer name, election type, the official voting method, seats, access mode, the three privacy options of §4.8, `max_ring_size`, `initial_electoral_roll_size` (the roll size at the opening — voters may still be added while the vote is open, §4.1, and the closure roll then shows the final size, so the *number* of additions is public), the ballot *parameters* (from which tally constraints are deterministically derived — the constraints themselves are never stored), and the candidates sorted by label.

The `method` field names the method whose ranking is the official result. It is taken from a closed list — `Schulze Winning`, `Schulze Margin`, `Ranked Pairs Winning`, `Ranked Pairs Margin`, `Copeland`, `Dodgson Quick`, `FPTP` — each value being the exact name the tally library resolves, and it is bound to the election type (`FPTP` for a single-choice election, a Condorcet method otherwise). Being part of block 0, the method is committed before the first vote: it cannot be chosen once the ballots are known.

```
{
  "protocol_version": 1,
  "type": "configuration",
  "election_id": "0c2e7e2a-...",
  "block_number": 0,
  "previous_hash": null,
  "title": "Best lake of the Alps",
  "organizer_name": "Club Alpin",
  "election_type": "preferential",
  "method": "Schulze Winning",
  "seats": 1,
  "voter_access_mode": "closed_nominative_list",
  "enrolment_mandatory": false,
  "non_anonymous_voting": false,
  "nominative_participation": false,
```

```
  "weighted_voting": false,
  "max_ring_size": 600,
  "initial_electoral_roll_size": 55,
  "ballot_rules": {
    "blank_allowed": true,
    "full_ranking_required": false,
    "partial_ranking_allowed": true,
    "ties_allowed": false
  },
  "candidates": [
    { "label": "Annecy" },
    { "label": "Bourget" },
    { "label": "Leman" }
  ]
}
```

The schema itself enforces the one-way dependency `non_anonymous_voting = true` ⇒ `nominative_participation = true` — a configuration violating it is never sealed. Purely cosmetic presentation metadata (logos, photos, thank-you message) is *not* in the block, not hashed, and not part of the verified definition.

`rings` — block 1, then one per emergency ring. Each ring's complete public-key list, keys in prefixed form, sorted ascending; rings sorted by ring id. Whether a `rings` block is a launch or an emergency block is deducible from its position — no extra flag:

```
{
  "protocol_version": 1,
  "type": "rings",
  "election_id": "0c2e7e2a-...",
  "block_number": 1,
  "previous_hash": "3fa9...e2c1",
  "rings": [
    { "ring_id": "a1b2...-uuid", "public_keys": ["pk_3f..._9a1b2c3d", "pk_7c..."] }
  ]
}
```

`votes` — batched ballot blocks. Ballots sorted by key image (chronology-free); each entry carries the ring id (so a verifier knows which key set to check against), the key image and signature in prefixed form, and the **verbatim signed payload as an opaque JSON string** — never re-serialized, so the verifier recovers the exact signed bytes by reading the string:

```
{
  "protocol_version": 1,
  "type": "votes",
  "election_id": "0c2e7e2a-...",
  "block_number": 7,
  "previous_hash": "bc41...11d9",
  "ballots": [
    {
      "ring_id": "a1b2...-uuid",
      "key_image": "ki_9d..._4e5f6a7b",
```

```
      "signature": "blsag_4e...",
      "payload": "{\"v\":1,\"election\":\"0c2e7e2a-...\",\"ballot\":\"A>B>C\",\"weight\":1}
    }
  ]
}
```


`closure` — the final block and chain terminator. No block may succeed it: possessing the signed closure block *is* the proof that a chain is complete. It carries the head totals and the **participation register** — the roll at the close, launch voters and mid-election additions alike, none marked — published here once, sorted by voter ID:


```
{
  "protocol_version": 1,
  "type": "closure",
  "election_id": "0c2e7e2a-...",
  "block_number": 42,
  "previous_hash": "77aa...03bc",
  "totals": {
    "block_count": 43,
    "ballot_count": 1300,
    "ring_count": 5,
    "voted_count": 1300,
    "voted_weight": 1300
  },
  "electoral_roll": [
    {
      "voter_id": "d4e5...-uuid",
      "key_provisioning_mode": "server_generated",
      "ring_id": "a1b2...-uuid",
      "distribution_count": 2,
      "has_voted": true,
      "weight": 1
    }
  ]
}
```


The participation register is the **denominator of the anti-stuffing bounds** (§5.11) and the substrate for per-voter fraud detection — without it, "no stuffing" would not be verifiable. Each line's `key_provisioning_mode` distinguishes server-generated, organizer-imported, and voter-enrolled keys, so the share of voters whose key the platform ever held is itself a public, per-election fact. When the nominative disclosure (§4.8) is on, lines of voters who voted additionally carry `voter_name` and `email_digest`; abstainers never do. At seal time the builder cross-checks every total it can prove from the block itself (`block_count`, `voted_count`, `voted_weight`) and refuses an internally inconsistent block.

A **cancelled** election emits no further blocks and no tombstone block: the chain simply stops without a closure block, which is itself the cryptographic evidence of cancellation for anyone holding the prefix.

5.6 Serialization: sign the file, not a canonical model

The format's foundational rule is:

> **stored bytes = served file = hashed bytes = signed bytes.**

Each block is serialized exactly **once**, at creation; those bytes are written to storage, served to the public verbatim (raw byte passthrough, never through a re-encoding JSON response layer), hashed for the chain link, and signed. Consequences and rules:

- **No canonical JSON (RFC 8785) is needed.** Canonicalization matters only when a verifier must *regenerate* bytes from a logical model and agree with the signer. Here the verifier hashes and verifies the very file it downloads, so formatting is irrelevant to integrity — the format deliberately chooses human-friendly, pretty-printed UTF-8 JSON, with each element of the one large collection (`candidates`, `rings`, `ballots`, `electoral_roll`) emitted as a single compact line.
- **Blocks are streamed, never materialized.** Large blocks (a closure roll can reach hundreds of megabytes) are written iteratively to a temporary file while the SHA3-512 is computed in-flux, and validated element by element against the item schema — constant memory on both the writing and the reading side (the read-side verifier is a streaming parser as well).
- **The ballot payload stays an opaque string** inside `votes` blocks and is never re-serialized (§5.5).
- **Duplicate JSON keys are rejected outright**, on both sides. Parsers disagree on duplicate keys (first-wins vs last-wins), so a duplicate-key block could present two different ballot sets to two verifiers while hashing and verifying identically for both — a forgery vector closed by a streaming lexer pass before any content is trusted.
- **All numbers are non-negative integers**; no floats appear in any block.
- **Content ordering is deterministic and content-derived**: candidates by label, public keys ascending, ballots by key image, roll lines by voter ID — always in byte order (`memcmp`), never a linguistic collation; the builder refuses unordered content at seal time.

5.7 Sealing and anchoring: how a block is produced

Block production is asynchronous (queued, per-election-serialized jobs) and follows a fixed pipeline whose committed database row is the arbiter:

1. **Build** the logical block from current data; **serialize** it once, streaming, into a temporary file.
2. **Validate** during sealing: schema conformance of the envelope and of every streamed item (a malformed block is never hashed or signed), plus chain coherence — `previous_hash` equals the current head's stored content hash and `block_number` equals head + 1.
3. **Hash and sign**: `content_hash = SHA3-512(bytes)` computed in-flux; a detached OpenPGP signature is taken over the exact same bytes (§5.8).
4. **Write** the block file and its `.sig` sidecar to object storage, keyed by (election, block number). Content is deterministic, so a retried write is idempotent.
5. **Commit the anchor**: one database transaction inserts the append-only anchor row (content hash + signature + sequence) and flips the batched ballots' `is_chain_published` flags. A uniqueness constraint on (election, block number) makes forked numbering impossible — a losing concurrent writer fails and retries against the new head.

Validity is *location-independent*: a block is valid iff its bytes hash to the committed content

hash and the signature verifies. A committed row whose object is missing is detectable and re-derivable; an orphaned object with no row is inert. A failed close-time flush is retried by a permanent catch-up sweep until the closure block exists — a delay, never an amputated chain.

The public consumes plain files — no API required: a per-election manifest listing every block (number, type, hash, signature timestamp, artifact URLs and OpenTimestamps status), the `{election}_{n}.json`, `{election}_{n}.json.sig` and `{election}_{n}.json.ots` artifacts — each named with its election id — served from object storage behind a CDN, and the platform's signing certificates at a well-known TLS endpoint. Everything is cacheable and mirrorable by design.

5.8 Platform signature and key management

Each block is signed with a **detached OpenPGP v6 (RFC 9580) signature, Ed25519**, produced by a pinned, vendored build of the Sequoia `sq` CLI. A third party verifies a block with one stock command:

```
$ sq verify --signer-file signing-keys.pgp \
    --signature-file 3.json.sig 3.json
```

Design points, stated factually:

- The signature covers the file bytes themselves. The signature's creation time is a hashed subpacket; a notation subpacket (`domain@condorcet.vote = chain-block/v1`) domain-separates chain-block signatures from any other artifact class the platform may sign later. A stock verifier that ignores the notation still verifies the block.
- Signing keys live in an ordered key set (last entry active, earlier entries retired-but-verifiable). Rotation stamps the outgoing key with an expiration self-signature at the rotation instant, so any RFC 9580-compliant verifier rejects material a retired key signs later, while everything it signed inside its window keeps verifying. The complete certificate set is published at a well-known TLS endpoint.
- This is containment hygiene, not compromise immunity: a *leaked* key can back-date signatures inside its historical window — a window the per-block OpenTimestamps anchor bounds (§5.9). Genuine revocation remains documented future work.
- V6 signatures require an RFC 9580-capable verifier (Sequoia, rpgp, gopenpgp, recent OpenPGP.js); GnuPG cannot verify them today. The published verification instructions name Sequoia.

5.9 External time anchor: OpenTimestamps

Every block is additionally anchored in **Bitcoin via OpenTimestamps**. At sealing, right after the block's database anchor commits, the platform submits the block file's **plain SHA-256** to the public calendar servers — deliberately without the usual OTS privacy nonce, since the bytes are public by design — so the calendar commitment equals `sha256({n}.json)` and any stock OpenTimestamps tooling verifies the proof against the downloaded block with no trust in the platform:

```
$ ots verify -f 3.json 3.json.ots
```

The proof lives as a third sidecar, `{election}_{n}.json.ots`, downloadable alongside the block and its signature whatever its state: a *pending* proof already carries the calendar commitments; once Bitcoin confirms — typically within hours — an hourly reconciliation collects

the attestation, replaces the stored sidecar with the enriched proof and flips the block's published OTS status from `pending` to `completed` (the manifest exposes both the proof URL and this status). Submission is strictly best-effort: a calendar outage never delays or fails block publication, and the same hourly sweep re-submits any proof lost to one.

What the anchor adds: the signature's creation time is the platform's own claim, but a Bitcoin attestation is not — it proves the block's exact bytes existed **no later than the attested Bitcoin block**, independently of the platform's keys. This bounds the back-dating window of a leaked signing key (§5.8) and strengthens the mirroring model (§5.10) with third-party proof of existence. The platform reports what a proof claims and performs no on-chain verification itself; verifiers do that with standard tooling, as above.

5.10 What "immutable" means here — and what it does not

The platform signature proves **authenticity** (the platform produced a block); on its own it does not prove **immutability**, because the platform holds the signing key and could re-sign a rewritten history. Immutability is achieved *in practice* through **independent mirroring**: every block is a plain, cacheable, mirrorable file, and any observer who retained an earlier chain head can oppose its signed copies to a rewrite, making tampering evident. Absent any witness, a malicious platform could re-issue a validly-signed alternate history undetected. This is the trusted-third-party model made precise — trusted at vote time, auditable afterwards *to the extent the public actually retained copies* — and it is stated here rather than glossed over. Protocol version 1 does add one external anchor: every block carries an OpenTimestamps proof of existence in Bitcoin (§5.9). The anchor proves *when* bytes existed — a rewritten history cannot obtain back-dated attestations — but it does not by itself prove a given history is the only one published; independent mirroring remains the rewrite defence.

5.11 What anyone can verify

From the published artifacts alone, with no trust in the platform:

1. **Chain integrity** — every block's signature verifies against the published certificate set; each `previous_hash` equals the SHA3-512 of the previous file (`openssl dgst -sha3-512 block-k.json`); numbering is consecutive from 0; every block carries the same election UUID; exactly one closure block terminates the chain; every block validates against its JSON Schema and contains no duplicate keys. Each block's OpenTimestamps proof additionally verifies against its file — existence no later than the attested Bitcoin block (§5.9).
2. **Ballot authenticity** — `verify_vote` passes for every published ballot against its ring's published keys.
3. **No same-key reuse** — all key images are unique within the election.
4. **No per-ring stuffing** — for each ring, published ballots equal the number of voters whose participation line names that ring.
5. **The binding global bound** — total ballots $\leq$ total voted participation lines; in weighted elections, the multiset of signed ballot weights equals the multiset of voted-line weights. Stuffing within these bounds forces fabricated participation lines, each detectable by the voter concerned (§6.2).
6. **The tally** — re-serialize the published payloads to a CEF document, derive the vote constraints from the published ballot parameters by the documented rule, and recompute with the referenced engine version (or any CEF-compliant implementation); the

blank/null classification and the final ranking reproduce exactly. The elected set is the top *seats* positions of that ranking.

Each voter can additionally verify their own line (no fabricated activity) and locate their own ballot via their receipt (§4.7) — including before batch publication, through a rate-limited exact-match lookup keyed by the key image, which is a high-entropy value only a receipt holder can present.

5.12 Tally reproducibility

Tallying is performed by the open-source CondorcetPHP engine through a typed API, with the method committed in the configuration block (§5.5); the published result records the engine contract version and the library version. For a Condorcet election the platform also computes the other methods of its list in the same pass and shows their rankings for information; only the committed method's ranking is official. The CEF document offered for download is a convenience, never an input — it is reproducible from the register. Ballot classification (*expressed*, *blank*, *null* — the latter split into null-by-format and null-by-constraint) is a deterministic, published rule over the published ballot parameters, so independent re-tallies reproduce the exact counts. The engine's response is not itself signed: the integrity guarantee is not the engine but the reproducibility of the computation from signed, published inputs — a divergence between tally implementations would be a software bug, not a verifiability failure, because the fixed signed vote list is the guarantee.

6 Threat model

6.1 Defended threats

Threat	Defense
Linking a ballot to a voter (public, organizer, later logical DB export)	Ring signature; no stored voter-to-key or voter-to-ballot correspondence; chronology-free storage and publication
Double voting with the same key (incl. replay)	Key-image uniqueness, database constraint, publicly checkable
Double voting with different keys	Participation register: guarded `has_voted` flip plus the distribution cap, both transactional
Ballot stuffing by the platform	Per-ring and global ballot/participation bounds (§5.11); stuffing forces fabricated participation lines, detectable by the affected voters
Ring draining / cornering a voter into a sparse ring	Distribution cap, 1.5x provisioning, deficit-weighted assignment, early emergency rings
Chronological traceability of ballots	Random UUIDs, no timestamps, batch publication in key-image order

Threat	Defense
Tampering with published history	Hash chain + platform signatures + independent mirroring (tamper-evident to any witness)
Forged or ambiguous block content	Schema validation + duplicate-key rejection on both sides; deterministic content ordering
Cross-election correlation of a reused key	Election-scoped key images (DDH)

6.2 Accepted limitations — documented, not hidden

- **An actively malicious platform at vote time.** The platform transiently knows voter-to-key at distribution (server-generated mode) and voter-to-ballot at submission (all modes). In server-generated mode a malicious platform could precompute the key image of every key it minted and thereby deanonymize every ballot in the register. Imported and enrolled keys remove both this precomputation power and any possibility of the platform signing in the voter's stead; BYO-only elections remove server-held secrets entirely. This bounded trust *is* the model's central pillar, and the reason the BYO modes exist.
- **Coercion and vote-selling.** Receipts are provable, and independently of receipts, published cleartext rankings enable pattern ("Italian") attacks: a coercer mandates a recognizable low-salience ordering and checks the register for it. This affects every voter regardless of key mode, needs no voter-to-ballot link, and is the price of publishing verifiable cleartext ballots. The protocol therefore targets private and associative elections, not coercion-sensitive public ones.
- **Fabricated participation for no-shows.** The platform could mark an absent voter as having voted and stuff a ballot within the published bounds; only the affected voter can detect it, by checking their own register line. The opt-in nominative disclosure partially counters this by letting an auditor holding the true email list recompute every digest. The same bound covers fictitious voters added while the vote is open (§4.1): the trust already placed in whoever composes the roll, shifted in time — the genesis block publishes the opening roll size, so the number of additions is public, never their identity.
- **Sparse-ring degeneracy.** A ring whose *participants* number one exposes that ballot's voter ID publicly (a pseudonym, not a name — resolving it to a person requires private platform or organizer knowledge, unless the nominative disclosure is on). Mitigation is structural (assignment formula, early emergency rings, even BYO partitioning) but the residual — a last ring below the floor at close — is inherent to any verifiable election with a public participation register and is accepted; such ballots are still counted. With mid-election additions an emergency ring's members are mostly added voters, so having voted in one reads as "joined after the opening" — a pseudonym-level inference, never the ballot.
- **Physical-storage side channels.** PostgreSQL stamps tuples with the writing transaction id (`xmin`), which momentarily creates a deterministic physical join between a ballot and its participation flip (one transaction), invisible in logical exports but readable with raw physical access. A twice-daily scrub rewrites stamped participation tuples under a single mute transaction and runs deterministic vacuums so dead versions carrying original stamps are removed; a fresh stamp survives at most one scrub cadence. Archived WAL and web access logs remain an infrastructure/retention concern, consistent with the

trusted-at-vote-time model.

- **Client-side compromise during the signing window.** The voting page is standalone, framework-free, and served under the platform's strictest first-party-only CSP; an XSS during the signing window remains the residual client-side risk this policy minimizes but cannot eliminate.
- **Voter contact data at rest.** The platform always holds voter names and emails in clear (VoteLink delivery is email-only, and the organizer needs a searchable roll), so a database compromise exposes them directly — an accepted, documented widening of the platform-side voter-to-email link. Voting secret keys, by contrast, are always encrypted at rest and deleted at distribution.

7 Implementation safeguards

Beyond the cryptography, a small number of implementation principles carry protocol guarantees directly. They are stated here at the level of principles; platform operations at large are outside this document's scope.

7.1 Protecting the signing window

The voting page is a standalone, framework-free page under the platform's strictest Content-Security-Policy: first-party only, no third-party scripts, no analytics or tracking anywhere, WASM execution enabled, form posts disabled in favor of fetch-only submission. Built assets carry subresource integrity hashes (SHA-384), so a compromised CDN edge cannot silently alter the signing code path. The voting flow lives on its own subdomain (`voting.condorcet.vote`) with host-only cookies, isolating its session state from the rest of the site — and **no intermediate CDN sits between the voter's browser and the platform for voting data**: ring payloads, key distribution, and ballot submission travel directly to the origin over TLS (only static built assets are edge-cached).

7.2 Data minimization and chronology hygiene

The platform stores nothing the protocol does not need and publishes nothing that dates or identifies a ballot: random UUIDv4 identifiers on the public surface, no ballot timestamps at all, verbatim ballot bytes as the single source of truth, secrets encrypted at rest and actively deleted at their use-by point (distribution, vote, close), transient in-memory secrets scrubbed via explicit memory-zeroing on both the PHP side (`sodium_memzero`-backed value objects) and the Rust/WASM side (zeroized buffers), and the physical-chronology scrub of §6.2 running as a scheduled background invariant.

7.3 The database as the single arbiter

Every user-facing invariant — distribution caps, the `has_voted` flip, key claims, block numbering — is enforced by guarded conditional writes and unique constraints in PostgreSQL, never by in-process state: correct under multi-worker, multi-instance execution by construction, with no lock ordering and no deadlocks. Signature verification (pure CPU over immutable public data) runs *before* the write transaction, so the crypto never holds a lock. Ballots and chain anchors are append-only by trigger; block production is serialized per election with a uniqueness constraint as the hard backstop against forks.

7.4 Publication isolation

Chain artifacts live on their own object-storage bucket with their own credentials, separate from any other stored content and fronted by its own CDN hostname — a defect in any upload path can never write into the chain bucket. The bucket additionally enables the cloud provider's **object-lock (write-once) mechanism**: once written, a block file can be neither overwritten nor deleted, even with the platform's own storage credentials — a storage-level enforcement of the append-only rule, complementing the database triggers and the hash chain. Blocks are served as raw bytes; no framework layer can reformat them.

7.5 Engineering discipline

The test suite exercises the **real WASM cryptography** — never mocks — on both the server (Extism) and browser (wasm-bindgen) paths, including full browser-driven ballot journeys; the crate itself pins frozen protocol vectors (any silent change to a domain string, binding format, or byte layout fails the build), runs bit-level malleability sweeps and fuzz targets over all parsers, and audits dependencies in CI. Specifications are versioned in-repo and kept in lockstep with code by explicit convention; this whitepaper is downstream of both.

8 Design trade-offs: advanced cryptography versus usability

The protocol repeatedly chooses the *simplest mechanism that keeps the guarantee checkable*, and accepts documented costs in exchange. The decisions below are presented factually — each is a trade, not a free lunch.

8.1 Verifiability by publication, not by advanced tally cryptography

Mix-nets, homomorphic tallies, and zero-knowledge tally proofs were considered and rejected as disproportionate for the target use cases. Instead, verifiability comes from publishing signed cleartext ballots and recomputing the tally deterministically. **Gained:** anyone can audit with `sq`, `openssl`, and any CEF-compliant tallier; no trusted ceremony; no esoteric verifier software. **Cost:** published rankings enable pattern attacks (§6.2) and preclude coercion resistance; the platform must be trusted at vote time.

8.2 Receipts over receipt-freeness

Individual verifiability (each voter can find and check their own ballot) was chosen over receipt-freeness; the two are mutually exclusive without substantially heavier machinery. **Gained:** every voter is a potential fraud detector, with zero added interaction cost. **Cost:** receipts are provable and transferable (§3.4), so vote-selling and coercion cannot be cryptographically prevented — hence the explicit positioning on private elections.

8.3 Server-generated keys by default, BYO-only as the trust-minimizing mode

Requiring every voter to generate and safeguard a key pair would shift the main failure mode onto the least technical participants. The default therefore lets the platform mint and distribute keys on demand — zero friction, at the cost of trusting the platform at distribution time (including its theoretical key-image precomputation power, §6.2). The trust dial is per-voter (any voter can import or enrol their own key, with no platform-side fallback) and goes all

the way to **BYO-only mode**, a one-setting election option that removes server-held secrets entirely and, with them, most of the distribution-side trade-offs of this section — at the price of shifting key custody onto every voter (see the comparison table in §4.2). The choice between the two is per election, not a platform constant.

8.4 Account-free voting through emailed capability links

Voters never create accounts or passwords; their VoteLink is a bearer capability delivered by email. **Gained:** the single highest-impact usability decision in the system; no credential lifecycle for voters. **Cost:** the platform necessarily holds every voter's email in clear, and possession of the link is possession of the vote (mitigated by hashing the token at rest, session scoping, and rate limiting; accepted within the model).

8.5 A capped ring size: one ceiling, one launch size per election

Rings are capped at 600 keys per election and every launch ring of an election shares one size under that cap, derived from the frozen roll (§4.3) — rather than per-voter rings or fully dynamic sizing. At the cap a ballot costs about 19 KB of signature and about half a second of signing and of verification; a small election on the 100-key floor costs a sixth of that. The cryptographic anonymity set is bounded to the ring rather than the whole electorate — but the *effective* anonymity set is the ring's participating voters, which padding never enlarges: for any election that fits in a single ring (up to 400 voters) it is the whole participating electorate, at 100 keys exactly as at 600. **Gained:** uniformity (all launch rings of an election are alike, all emergency rings are the full cap), resistance to ring-draining (capacity always covers three distributions per server-provisioned voter under the cap), simple public verification, and predictable performance on modest devices. The 600 cap matches the publication batch floor — one polling station's worth of voters — and is also the size of every emergency ring, so that a roll growing mid-election is absorbed by a few large rings rather than fragmented into many small ones.

8.6 The distribution re-delivery cache

Strictly, the protocol wants the voter-to-key correspondence to never exist outside the voter's device. In practice, voters reload pages and lose tabs; without re-delivery, each reload would burn one of three capped distributions. The platform therefore keeps the distributed secret, encrypted and keyed to the voter, for at most 180 minutes and deletes it on vote — a deliberate, TTL-bounded widening of the trusted-at-vote-time window, traded for not locking real voters out of legitimate votes. BYO-only elections eliminate this trade-off outright: with no distribution, the cache — and the transient binding it represents — never exists.

8.7 A central platform with mirrorable evidence, not a decentralized ledger

Running a decentralized ledger would add operational and conceptual weight for the organizers this platform targets. Instead the chain is centrally produced but designed for **cheap independent custody**: plain JSON files, one stock verification command, no API needed — plus a per-block OpenTimestamps anchor (§5.9) that adds third-party proof of existence at negligible cost. The immutability guarantee is exactly as strong as the mirroring it receives (§5.10) — stated as a property, not marketed as equivalent to consensus.

9 Known limitations and planned work

- **Independent review.** An external review of the protocol and its implementation is planned — with particular attention to the election-scoped key-image construction, a deliberate, narrow deviation from the published bLSAG construction. Its report will be linked from the platform's security page.
- **eIDAS-grade time anchoring.** Per-block OpenTimestamps anchoring is implemented (§5.9); an RFC 3161 TSA could complement it where eIDAS-grade probative value is required.
- **Genuine key revocation.** V1 relies on expiration-at-rotation; hard-revocation semantics (with out-of-band revocation certificates) are sketched for a later version — the per-block OpenTimestamps anchor (§5.9) provides the external anchoring they were contingent on.
- **Retention and archival policy.** Long-term retention of registers and operational metadata (GDPR alignment) is intentionally deferred and tracked in the specifications' open questions.

10 References

- [LWW04] J. K. Liu, V. K. Wei, D. S. Wong. *Linkable Spontaneous Anonymous Group Signature for Ad Hoc Groups.* IACR ePrint 2004/027.
- [ZtM20] koe, K. M. Alonso, S. Noether. *Zero to Monero* (2nd ed., 2020) — the bLSAG construction and its linkability analysis.
- [CP92] D. Chaum, T. P. Pedersen. *Wallet Databases with Observers.* CRYPTO '92.
- [FS86] A. Fiat, A. Shamir. *How to Prove Yourself: Practical Solutions to Identification and Signature Problems.* CRYPTO '86.
- [Ristretto] The Ristretto Group. https://ristretto.group — prime-order group over Curve25519; implemented by `curve25519-dalek`.
- [RFC7693] M-J. Saarinen, J-P. Aumasson. *The BLAKE2 Cryptographic Hash and MAC.* RFC 7693.
- [BLAKE3] J. O'Connor, J-P. Aumasson, S. Neves, Z. Wilcox-O'Hearn. *BLAKE3: one function, fast everywhere.* 2020.
- [FIPS202] NIST. *SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions.* FIPS PUB 202.
- [RFC9580] P. Wouters et al. *OpenPGP.* RFC 9580 (OpenPGP v6).
- [RFC8032] S. Josefsson, I. Liusvaara. *Edwards-Curve Digital Signature Algorithm (EdDSA).* RFC 8032.
- [cryptoVote] The `crypto_vote` Rust crate. https://github.com/CondorcetVote/cryptoVote
- [Condorcet] J. Boudry. *Condorcet* — the CondorcetPHP engine and the Condorcet Election Format (CEF). https://github.com/julien-boudry/Condorcet
- [Extism] Extism — the WebAssembly plug-in framework used for server-side verification. https://extism.org
- [Sequoia] Sequoia-PGP — the OpenPGP implementation whose `sq` CLI signs and verifies chain blocks. https://sequoia-pgp.org

---

Publisher. Condorcet.Vote is operated by **Condorcet.Vote SASU**, a société par actions simplifiée under French law — SIREN 000 000 000, RCS [city] *(identification placeholders — to be completed before publication).*