Ember: A Bitcoin-Anchored Proof-of-Stake Consensus Reducer

Jeremy Rubin
jeremy.l.rubin [at] gmail [dot] com

Ember is a proof-of-stake consensus reducer for Bitcoin-anchored bonded identities. It converts asynchronous, ballot-numbered attestations from anchor-scoped per-domain chains into canonical decision outcomes for domain-scoped ballot streams. Attestations are reduced locally into decision or impossibility certificates rather than re-attested over the network. Signed genesis bindings bootstrap per-domain chains, and deterministic Purify nonce proofs eliminate any need to pre-register nonces. This paper specifies Ember's state and message model, endorsement semantics, deterministic reduction rules, sampling and leadership interfaces, and fault-handling behavior under reorgs and equivocation. The protocol objective is twofold: safety (conflicting outcomes for the same ballot are not both certifiable without slashable supermajority misbehavior) and practical liveness (ballots eventually resolve when sufficient online stake is available and network conditions satisfy the stated assumptions). This document is implementation-oriented and consensus-complete; system integration and economics are intentionally left to companion papers.

CONFIDENTIAL

This document is confidential and not for general distribution.

The information herein is proprietary and intended solely for authorised recipients. Unauthorised copying, distribution, or disclosure is strictly prohibited.

Nothing in this document constitutes a commitment to future actions, financial returns, or product features. All specifications are preliminary and subject to change without notice.

No legal, tax, or investment advice. This publication is provided for informational purposes only and does not constitute an offer to sell, or the solicitation of an offer to buy, any security or financial instrument. Recipients should obtain their own independent advice.

This document may contain **forward-looking statements**. Actual results may differ materially due to risks and uncertainties beyond Judica's control.

All information is provided **"as is"**, without warranty of any kind, express or implied. Judica disclaims all liability arising from reliance on, or use of, this material.

Certain techniques described herein are **patented** or **patent pending**. No licence to any intellectual-property rights is granted or implied.

Table of Contents

Contents

1 Introduction

Distributed services anchored to Bitcoin [15] need more than fraud proofs: they need a deterministic way to decide which proposed inputs become canonical, in what order, and under which validator eligibility view. Ember addresses this problem as a consensus reducer for bonded operators. Each bonded identity contributes signed attestations in an anchor-scoped namespace; Ember interprets those attestations as weighted endorsements and reduces them into certified ballot outcomes. Conceptually, Ember borrows accountable quorum reasoning from BFT replication [7, 22] and economically weighted participation from proof-of-stake protocols [11, 6], but applies them to anchor-scoped attestation streams rather than to a single replicated log.

The design targets a setting where participation is economically accountable and where chain reorgs are a routine environmental condition rather than an exceptional failure mode. Operators are represented by stake-bearing anchor + stake constructions, messages are hash-linked and replayable, and every decision artifact is tied to an explicit evidence view. This keeps validation deterministic across implementations and makes disputes auditable post hoc.

Ember's primary engineering goals are:

- **Deterministic reduction**: honest nodes that observe the same admissible attestation set derive the same decision roll.
- **Accountable safety**: conflicting certifications imply slashable behavior under the protocol's signing and endorsement rules.
- **Operational liveness**: under stated synchrony and online stake assumptions, ballots progress without centralized coordination.
- **Implementation clarity**: node behavior under reorgs, delayed delivery, and partial visibility is specified at a level suitable for production code and interoperability testing.

This paper is intentionally consensus-only. It does not define application-level business logic, or enforcement-market structure except where those concerns directly constrain consensus correctness.

2 Scope and Reader Model

This paper is intended for protocol implementers, auditors, formal reviewers, and node operators. It specifies the precise PoS reduction mechanism, admissibility rules, and safety/liveness assumptions required to implement Ember consistently across independent clients.

3 Model and Terminology

3.1 Notation Aid

- χ: domain identifier (application or shard).
- b: ballot number (index) within domain χ.
- v_b^{χ}: value filling ballot b in domain χ.
- s_i: anchor-scoped consensus identity with active stake weight $\omega_i(B)$.
- a_i: anchor identifier (`AnchorId`), equal to the anchor transaction txid.
- $c_{i,\chi}$: attestation-chain identifier for anchor a_i and domain χ.
- $g_{i,\chi}$: signed genesis attestation for chain $c_{i,\chi}$, carrying the binding $(a_i, \chi, b^{\chi}_{\text{start},i})$.
- $b^{\chi}_{\text{start},i}$: first ballot number on which chain $c_{i,\chi}$ may emit a payload-bearing attestation. If $b^{\chi}_{\text{start},i} > 1$, the genesis binding indicates that, from identity s_i's local view at bootstrap, every ballot $b < b^{\chi}_{\text{start},i}$ in domain χ is already settled.
- When domain index χ is omitted from $\mu^{(b)}_{i,\chi}$, $\Sigma^{(b)}_{i,\chi}$, or $E^{(b)}_{i,\chi}$, the statement is for a fixed chain/domain.
- $\omega_i(B)$: active stake weight represented by identity s_i at Bitcoin view block B.
- $\Omega(B)$: total active stake in the electorate as of block B.
- $\mu^{(b)}_{i,\chi}$: payload-bearing attestation from identity s_i on chain $c_{i,\chi}$ for ballot b.
- $E^{(b)}_{i,\chi}$: set of endorsements (votes) by s_i in domain χ at ballot b.
- $\mathcal{C}$: set of all domains.

- $\mathcal{I}$: set of consensus identities, indexed by anchor identifiers (`AnchorId`); $\mathcal{U}$: set of stake outpoints (`StakeId`).
- $\mathcal{A}(B) \subseteq \mathcal{I}$: active identity set at block B.
- Θ: endorsement threshold fraction (typically $\frac{2}{3}$).
- $H(\cdot)$: cryptographic hash function
- $\Sigma_{i,\chi}^{(b)}$: attestation signature package (BIP340 signature plus Purify nonce proof) by s_i on $\mu_{i,\chi}^{(b)}$.

3.2 Terminology and Roles

- **Bond**: The logical staking identity composed of an anchor transaction plus zero or more associated stake outpoints.
- **AnchorId**: The txid of the anchor transaction.
- **AttestationChainId**: The logical chain identifier. It is a deterministic tagged hash of $(\texttt{AnchorId}, \chi)$; there is no separate registry chain.
- **Genesis Binding**: The repeated per-chain tuple is $(\texttt{AnchorId}, \chi, b_{\text{start}}^{\chi})$. Here b_{start}^{χ} denotes the first payload-bearing ballot for that chain. If $b_{\text{start}}^{\chi} > 1$, the binding indicates that earlier ballots in χ are already settled from that chain's bootstrap perspective.
- **Genesis Attestation**: The empty-payload root of a domain chain. Its attestation ballot field is 0 and is ignored for ordinary decision-making; the first payload-bearing attestation uses b_{start}^{χ} from the genesis binding.
- **Domain Chain**: A per-domain attestation chain under one anchor namespace; signed by the anchor key and authenticated by Purify nonce proofs.
- **StakeId**: A specific stake outpoint, represented as `txid:vout`, bound to an `AnchorId`.
- **Bond Seed Key**: Root secret from which the anchor signing key and the Purify key pair may be derived through isolated derivation branches.
- **Bond Anchor Key**: BIP340 key controlling the quick-break output and attestation signatures.
- **Purify Key**: Auxiliary key used only to prove that an attestation nonce was deterministically prepared for a specific topic (χ, b), including the genesis case $T(\chi, 0)$.
- **Operator**: Implementation-level actor controlling keys. This is not a protocol identity; one operator may control multiple bonds.
- **Referendum / Vote**: A proposal from a parliament member; on the wire the vote leaf carries raw domain payload bytes, while the addressed ballot number lives in the enclosing attestation.
- **Decision Roll**: Certificate that commits to a specific quorum (explicit decision, hang, or cloture/timeout). It is derived and stored locally rather than attested over the network.
- **Active Set / Electorate**: All active identities in scope for a given view block.
- **Bond lifecycle**: Pending (created, not yet active), Active (live and slashable), Slashed (penalized), End-of-Life (left the active set), Redeemed (collateral returned), Renewed (rolled forward), Reported (quick break spent, public invalidation).
- **Zeitgeist**: The sampling and attestations keyed to a particular block hash. This is not a formal epoch, as there is no fixed time period as a successor block propagates. Zeitgeist of A can generally be thought of, with a perfectly synchronous network, as the period between blocks A and B.
- **Domain**: Logical application/shard addressed within the protocol.
- **Ballot Number**: The monotonically increasing index within a domain's stream on which referenda are proposed and decided.
- **Attestation**: Signed structured data linked to a specific identity and its prior attestation history, emitted at times specified in the protocol.

[1]

- **Parliament**: Sampled subset of the electorate for a Domain during a Zeitgeist; members are ordered (queue) and one is the *recognized* proposer.
- **Ember**: The consensus reducer protocol that orders and finalises ballots from attestation chains.
- **Char Network**: A deployment context in which Ember runs.

3.3 Detailed Representation of the Deployment Model

This section presents a detailed representation of the core components and processes in the deployment model used throughout this paper.

3.3.1 Bonds, Identities, and Stakes

Let there be an identity set $\mathcal{I} = \{s_1, s_2, \ldots, s_N\}$ and a stake-outpoint set $\mathcal{U}$. Each identity $s_i \in \mathcal{I}$ is indexed by an `AnchorId` a_i. For each domain χ, identity s_i may have a derived attestation chain $c_{i,\chi} = H_{\text{chain-id}}(a_i, \chi)$, rooted by the signed genesis attestation $g_{i,\chi}$ defined in Section 3.2. Each stake outpoint $\kappa \in \mathcal{U}$ is a `StakeId` (`txid:vout`) bound to one anchor identifier and carrying value ω_κ. The effective weight of identity s_i at view block B is $\omega_i(B)$, measured in Satoshis[2]., $\omega_i(B) \in \mathbb{N} \mid 0 < \omega_i(B) < 2^{51}$. For a fixed view block B, the total active stake in the electorate is:

$$\Omega(B) = \sum_{s_i \in \mathcal{A}(B)} \omega_i(B).$$

3.3.2 Domains and Ballot Numbers

The network supports multiple domains, each identified by a unique domain tag $\chi \in \mathcal{C}$, where $\mathcal{C}$ is the set of all domains. Each domain χ has an associated sequence of ballot numbers $\{0, 1, 2, \ldots\}$; ballot 0 is reserved for the empty genesis attestation and carries no ordinary decision value. Each positive ballot number $b \geq 1$ may be associated with a value v_b^χ. For a fixed chain $c_{i,\chi}$, the first payload-bearing attestation uses the genesis-declared $b^\chi_{\text{start},i}$, and each later payload-bearing attestation increments that chain's ballot number by one.

3.3.3 Referendums and Messages

Consensus identities propose values for ballot numbers in domains by issuing *attestation messages*. On the wire, a non-genesis attestation for chain $c_{i,\chi}$ is:

$$\mu_{i,\chi} = \big(c_{i,\chi},\ \text{prev},\ b,\ \mathbf{H}_B,\ P,\ G_{i,\chi},\ \Sigma^{(b)}_{i,\chi}\big)$$

where:

- prev is a commitment to the previous attestation in the chain.
- b is the ballot number addressed by the attestation. If prev $= H(g_{i,\chi})$, validation requires $b = b^\chi_{\text{start},i}$; otherwise b must be one greater than the predecessor's ballot number.
- $\mathbf{H}_B$ is the tip-first Bitcoin header vector that fast-forwards from prior attestation context to the current claimed tip.
- P is the attestation payload. In the reference implementation it is either empty (genesis only) or exactly one `ReferendumVote` leaf carrying raw domain-defined payload bytes.
- $G_{i,\chi} = (a_i, \chi, b^\chi_{\text{start},i})$ is the repeated genesis binding defined in Section 3.2; genesis itself uses ballot field 0, while the first payload-bearing attestation uses $b^\chi_{\text{start},i}$.
- $\Sigma^{(b)}_{i,\chi}$ is a BIP340 signature plus a Purify nonce proof. For ordinary attestations the proof is checked against the topic $T(\chi, b)$; genesis attestations use $T(\chi, 0)$.

3.3.4 Endorsements and References

A bonded identity endorses a proposal by emitting a valid payload-bearing attestation for that ballot. For identity s_i, domain χ, and ballot b:

$$E^{(b)}_{i,\chi} = \begin{cases} \{(b, v_b^\chi)\}, & \text{if } s_i \text{ publishes } v_b^\chi \text{ for } (\chi, b), \\ \varnothing, & \text{otherwise.} \end{cases}$$

[1] In the reference implementation, attestations *may* additionally include cross-links to recently observed messages from other oracles to aid robustness and discovery, but this is an engineering choice rather than a requirement of the Ember protocol.

[2] the individual atomic unit of Bitcoin, 2^{51} is the smallest power of two greater than the max number of individual units of bitcoin

The wire format has no second endorsement primitive beyond the payload-bearing attestation itself. Local storage may later derive roll objects that summarize many endorsements, but the voting act is always the attestation.

3.4 System and Adversary Model

Network. The P2P network is *eventually synchronous* [8]: there exists a (possibly unknown) global stabilisation time after which all messages sent between honest nodes are delivered within a bounded delay. Concretely, we assume that after this stabilisation point the Char P2P layer delivers honest envelopes within Δ_{gossip}, and Bitcoin blocks arrive within a coarse bound Δ_{btc} (used in Section 5.2). Temporary partitions and chain reorgs before or during stabilisation are tolerated, subject to the drift bounds specified later.

Adversary. Controls a fraction f_{adv} of active stake with $f_{\text{adv}} < 1 - \Theta = \frac{1}{3}$, can schedule its own messages arbitrarily, delay honest envelopes within the network budget, and adaptively exploit leaked keys from nonce reuse. Equivocation is allowed; leaked keys must make bonds slashable.

Randomness. Leader sampling uses the current Bitcoin block hash [15] and domain id as a public seed with bounded miner grinding. A VRF beacon [14], a commit-reveal beacon [21], or VDF-assisted randomness [3] could further reduce bias.

Cryptography. Schnorr signature security follows the BIP340 setting [17], together with hash preimage/collision resistance and Merkle collision resistance for Bamboo [13]. Purify topic proofs follow the MuSig-DN nonce-derivation lineage [16] and the single-signer Purify construction used here [10]; they bind valid attestation nonces to a domain/ballot topic, and nonce reuse remains extractable as in Appendix B.

3.5 Performance Goals and Metrics

Latency. Measure wall-clock time from Bamboo payload eligibility to a Decision Roll. Common-case budget is leader emission plus one envelope propagation ($\Delta_{\text{domain}} + \Delta_{\text{gossip}}$); null-ballot fallbacks add another Δ_{gossip}.

Throughput. Bounded by the 3 MB payload cap per attestation and the Bitcoin block cadence. Aggregate throughput scales with the number of online bonds and outbound bandwidth of char-relay links.

A deployment-level throughput envelope is:

$$\text{throughput} \lesssim \frac{N_{\text{online}} \cdot S_{\text{payload}}}{T_{\text{btc}}},$$

where N_{online} is the number of online signing identities, $S_{\text{payload}} \leq 3$ MB is the per-attestation Bamboo cap, and T_{btc} is average Bitcoin block interval.

Fault coverage. Safety/liveness target $f_{\text{honest}} \geq \Theta = \frac{2}{3}$ online with bounded $x_{\uparrow}$ and $x_{\downarrow}$; outside that envelope the system should pause rather than finalize conflicting values.

4 Ember Consensus Protocol Specification

4.1 Bonds and Attestations

Char uses an anchor-scoped attestation namespace. Each anchor a_i has zero or more per-domain chains $c_{i,\chi} = H_{\text{chain-id}}(a_i, \chi)$, each rooted by the signed genesis attestation and repeated genesis binding $G_{i,\chi}$ defined in Section 3.2. Deterministic Purify proofs [16, 10] remove any need for a separate nonce-registry chain. Voting weight and slashing attribution resolve by the anchor a_i, not by the derived domain-chain id.

Each attestation commits to its predecessor hash, ballot number, Bitcoin header vector, payload hash, and repeated genesis binding (Section 4.2). Signatures are verified against the anchor's BIP340 key [17] and must also carry a Purify proof for the addressed topic (χ, b), or $T(\chi, 0)$ in the genesis case. If an anchor signs two distinct attestations for the same topic, the reused nonce leaks the signing key and the bond becomes slashable.

4.1.1 Anti-Equivocation via Nonce Reuse

Each bond commits on-chain to two public keys: a Taproot output key [18] used for quick-break and BIP340 attestation signatures [17], and a 64-byte Purify public key stored in the anchor `OP_RETURN`. These

may be deterministically derived from a single bond seed key using isolated derivation branches so that compromise of the attestation signing key need not reveal the nonce-generation key.

For every non-genesis attestation on domain χ and ballot b, the signer prepares a nonce against the topic $T(\chi, b)$, and the verifier checks that the Purify proof recovers the same public nonce R that appears in the Schnorr signature. Genesis attestations use $T(\chi, 0)$. This removes nonce pre-registration while making direct same-topic conflicts slashable; predecessor links remain necessary for ordering, reorg validation, and ancestry-consistency checks.

4.1.2 Slashing Mechanisms and Bonded Funds

Once a leaked key is detected, bonds become spendable to burn. Contracts delay exits (CSV) to leave time for challenges. In the prototype, slash and quick-break spends burn to `OP_RETURN` only after nonce reuse is detected; there is no covenant enforcement yet, so the burn is best-effort signalling until alternative enforcement are available.

Penalty mechanisms: rationale Penalties serve to align rational and honest behaviour. In the absence of penalties, a rational operator might deviate from the prescribed protocol whenever doing so appears locally profitable. By making misbehaviour explicitly costly – for example through bond confiscation or dilution – a penalty scheme can make the rational choice coincide with the honest strategy, or induce would-be attackers to abstain from participation entirely.

Penalty mechanism alternatives and extensions The mechanisms below are optional design extensions and are not required by the reference implementation.

The slashing transaction template can be extended in several directions. These are not required for the core Char protocol, nor would their addition change the core security and safety properties of Char, but they illustrate how different economic and policy choices can be expressed in bond construction and how they might impact dishonest behaviour.

Partial penalties The slashing mechanism can be designed to allow partial penalties. For example, an oracle's stake can be gradually reduced upon each instance of misbehaviour, rather than being entirely confiscated at once. This provides a more nuanced approach to penalising oracles and can be implemented by reallocating a portion of the funds to a new staking contract with reduced value.

Redirecting slashed funds Instead of burning the slashed funds, they can be redirected to long-term holding addresses or charitable causes, ensuring that the value remains within the ecosystem or serves a beneficial purpose. For instance, funds could be sent to an annuity contract that becomes claimable after an extended period, making it impractical for the cheating oracle to benefit directly.

Covenants While covenants are required to implement some of these designs in a succinct and programmable way [20, 12], they are not strictly necessary. For example, the slashing transactions can be required to be published in a follow-up transaction data commitment (such as an inscription or an `OP_RETURN`), which can then be used to drive a specific slashing transaction on cheating, instead of a free-for-all. Naive approaches with covenants may also lead to undesirable dynamics: if the slashing transaction is a covenant that enables very rapid unbonding (for example, in a few hours), then the amount of staked funds may be reduced quickly in a "bank-run" scenario. This makes the network less attractive to AVSes, as they may be subject to rapid degradation of consensus. A more conservative approach is to use a "slow-unbonding" pattern that requires a longer time to unbond, which encourages stakers to plan the duration they intend to remain in the network and to maintain consensus.

4.1.3 Bond Construction and Lifecycle

Figure 1 sketches the life-cycle of a *Char bond* in the current Anchor + Stake design. Onboarding is a two-step process: first create an *anchor transaction*, then fund one or more *stake outputs* that are bound to that anchor. The attestation namespace is then built under that anchor as zero or more per-domain chains, each bootstrapped by a signed genesis attestation.

1. **Anchor transaction layout**. A wallet helper uses `Char::BondAnchorBuilder` to construct canonical anchor outputs:

(a) output 0: a 330-sat *quick-break* Taproot key-path output;

(b) output 1: an `OP_RETURN` containing exactly one 64-byte push (the bond's Purify public key).

Wallet coin selection may add a change output, but the anchor semantics are fully determined by outputs 0 and 1.

2. **Stake transaction layout**. The stake-funding RPC creates stake UTXOs in separate transactions. A stake script is derived as:

$$\langle P \rangle \ \mathsf{CHECKSIGVERIFY} \ < \tau > \ \mathsf{CSV},$$

where P is the anchor key and τ is one of the supported candidate durations. The script's internal key is `NUMS_H` TapTweaked by the anchor transaction id, so each stake output is cryptographically bound to a specific anchor. Multiple stake outputs may coexist for one anchor and contribute additive weight.

3. **Activation and scheduling**. The bond index tracks anchors and stake outpoints separately:

   - An anchor enters *active stage* after 6 blocks.
   - A stake outpoint contributes weight only after its own 6-block activation delay.
   - A stake outpoint stops contributing in the final 12 blocks before its maturity height, or immediately once spent.
   - Spending the anchor quick-break output marks the bond broken and removes it from leader selection.

   Selectability therefore requires both an unbroken active anchor and strictly positive active stake at the queried height.

4. **Chain namespace bootstrap and genesis**. For each domain χ, the implementation derives a domain chain id:

$$c_{i,\chi} = H_{\text{chain-id}}(a_i, \chi).$$

When a domain chain is first needed, the node signs a genesis attestation with empty payload and binding $(a_i, \chi, b^{\chi}_{\text{start},i})$, where $b^{\chi}_{\text{start},i}$ is the first payload-bearing ballot for that chain. Choosing $b^{\chi}_{\text{start},i} > 1$ indicates that, from that bond's local view, earlier ballots in χ are already settled when the chain is bootstrapped. The genesis attestation itself uses ballot field 0, which is ignored for ordinary decision-making; the first payload-bearing attestation must then use $b^{\chi}_{\text{start},i}$, and later attestations increment by one ballot each.

The worker implementation then splits pending payloads by domain and emits one attestation per (a_i, χ) domain chain. If a chain does not yet exist, its signed genesis is committed locally and gossiped before the first payload-bearing attestation.

5. **Topic-bound nonce proofs**. The protocol does not pre-commit nonces from one message to the next.[3] Each signature carries a Purify proof [16, 10] that deterministically prepares its nonce for the attestation topic: $T(\chi, b)$ for ordinary ballots and $T(\chi, 0)$ for genesis. This gives third parties an auditable check that a signer followed the topic-binding rule without pre-registering nonces, maintaining a nonce reservoir, or storing future nonces. Implementers may still propagate prepared Purify nonces and proofs in advance for performance or latency reasons, but doing so is an optimization rather than a protocol requirement; the current implementation is documented at [9].

6. **Validation**. `BondAnchorSpec::ValidateTx` / `GetBondAnchorTxView` validate anchor structure: output 0 must be a 330-sat Taproot quick-break output, and output 1 must be a strict single-push `OP_RETURN` carrying only the Purify public key (no trailing fields). Stake outputs are validated by script binding and outpoint tracking in `charbondindex`.

7. **Life-cycle transactions**. In the active implementation, lifecycle spends are:

   Quick-break one-input spend of anchor output 0 using key-path Schnorr with

[3] In a prior version of the protocol, before Purify deterministic nonces, nodes pre-registered future nonces on a separate nonce-registry chain and maintained nonce-reservoir state across messages.

SIGHASH_NONE and ANYONECANPAY, burning the value to an OP_RETURN.

Stake slash for each slashable stake outpoint, a CSV script-path spend via the stake-slash helper burns the stake value once the outpoint's duration condition is met.

Roll legacy roll helper APIs exist, but anchor-only bonds are intentionally handled via explicit new anchor + stake transactions rather than in-place roll transactions.

On nonce reuse, the attestation worker gossips a tombstone attestation and broadcasts the quick-break and stake-slash transactions with the leaked key.

4.1.4 Security intuition

- The key-path quick-break allows the staker to disable their bond for consideration early, or to signal that the staker has equivocated and provide notice to slash bound stake outputs.
- Slashability attaches to stake outpoints (not the anchor output). Any equivocation detected while slashable stake remains lets honest parties confiscate that stake (i.e., burn the funds to an OP_RETURN or send to a mining fee, see Section 4.1.2).
- Every ballot topic (χ, b) authorizes exactly one public nonce under the bond's Purify key. Two distinct valid signatures for the same domain and ballot therefore reuse that nonce and leak the key directly (Appendix B). Any direct same-topic witness is therefore position-independent once the conflicting topic is identified. Conflicting genesis attestations are the $b = 0$ case; a backward walk is needed only to localize a later-discovered ancestry violation against the repeated genesis binding.

4.1.5 Implementation Approach

In the reference implementation, staking contract logic is implemented directly in C++ for tight Bitcoin wallet integration and operational control.

The contracts governing bonded identities and attestation chains could be expressed in a higher-level transaction-template DSL such as Sapio, which captures spending paths and state transitions. Appendix A shows a simplified example for one bond. Integrating formalized templates remains future work.

4.2 Bamboo Values and Local Roll Objects

Bamboo remains the typed leaf/value container and mmap-backed storage layer used by Char, but the network no longer carries a multi-key Bamboo tree in every attestation. Each attestation commits to exactly one BambooValue and its hash. The pending store may stage many candidate values locally, but the signing worker splits them into one attestation per domain payload before signing.

At the wire level, the reference validator accepts only two payload states:

1. an empty value, used only for signed genesis attestations;
2. a single ReferendumVote leaf, whose serialized body contains raw domain-defined payload bytes.

The ballot number is not duplicated inside the ReferendumVote leaf; it is taken from the enclosing attestation's ballot field. Attempting to carry a DecisionRoll, an ImpossibleRoll, or a legacy registration leaf inside an attestation is invalid under the rules.

Decision and impossibility certificates still exist as protocol objects, but they are derived locally from observed attestations rather than re-attested over the network. A DecisionRoll commits to a winning payload hash, the reference Bitcoin block hash, and a proof set of supporting CheckableAttestations. An ImpossibleRoll analogously certifies that no payload can reach threshold in the queried block view. Both are stored in the local tabulation database and surfaced via RPC. Intuitively, a verifier reconstructs a decision by replaying the cited attestations under the same Bitcoin/stake view and checking that their endorsed weight reaches quorum for the recorded payload hash. A compact end-to-end example appears in Section 8.1.

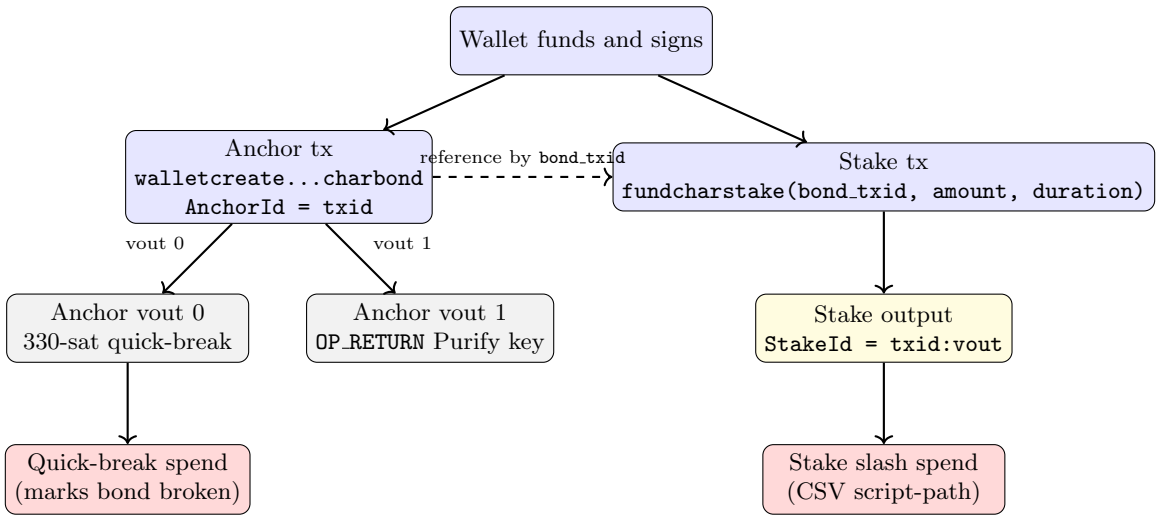

Figure 1: Life-cycle of a Char bond under the Anchor + Stake model. Stake funding is a separate transaction that references the anchor txid (`AnchorId`) as RPC input. The anchor transaction has a fixed structure with a quick-break output and a committed Purify public key; stake transactions create outputs with scripts that are cryptographically bound to the anchor. On detection of nonce reuse, the attestation worker gossips a tombstone attestation and broadcasts the quick-break and schedules stake-slash transactions using the leaked key when they become valid for spending. Derived domain-chain ids are deterministic hashes of the anchor id and domain id and are not shown.

4.3 Attestation Header Chains and Self-Synchronization

Each attestation carries a tip-first Bitcoin header vector

$$\mathbf{H} = (H_0, H_1, \ldots, H_{m-1}),$$

where H_0 is the claimed tip and each next element is one step older. Let $A = \text{prev}(H_{m-1})$ be the vector anchor hash. The vector is interpreted as a fast-forward bridge from an anchor in prior attestation history to a newer tip, including reorg moves across branches.

Lemma 1 (Accepted header vectors are contiguous and anchored)**.** *If `CheckReorgAttestations` accepts an attestation with non-empty header vector* $\mathbf{H}$*, then:*

1. *the vector is contiguous:* $\text{prev}(H_j) = \text{hash}(H_{j+1})$ *for all* $0 \le j < m - 1$*;*
2. *each* H_j *satisfies Bitcoin PoW validation (and contextual difficulty checks when height context is available);*
3. *the anchor* A *connects to prior attestation history for that chain (or the attestation is rejected as invalid/no-context).*

Proof. The validator checks vector contiguity by enforcing each header's `hashPrevBlock` against the expected predecessor while iterating the vector, checks PoW on every header, applies contextual difficulty transition checks when anchor-height context is known, and requires anchor connectivity against prior attestation history (otherwise returning `INVALID_HEADER` or `NOCONTEXT_ATTESTATION`). □

Theorem 1 (Header-chain self-synchronization safety)**.** *Fix one attestation chain and local prior history. If a new attestation is accepted by `CheckReorgAttestations`, then local header state either stays on the same tip (tip-only vector) or advances to a tip whose chainwork-from-anchor is strictly greater than the prior tip's chainwork-from-anchor. Direct sibling jumps without a valid bridge are not accepted.*

Proof. For unchanged tips, validation requires a tip-only vector. For tip changes, acceptance requires (i) contiguity and PoW-valid headers, (ii) anchor linkage to prior history, (iii) no overlapping reuse of already traversed history between anchor and prior tip, and (iv) strict increase of chainwork-delta from the common anchor. Therefore a stale node can self-synchronize by accepting a valid fast-forward vector, while malformed sibling jumps fail these checks. □

4.4 Consensus and Ordering

Bonded stake plus deterministic sampling yields a PoS-style reducer: two-thirds weighted signatures certify a state. Leaders are sampled per Zeitgeist/-

domain (Section 4.4.2). Equivocators can be slashed on-chain. Each attestation commits to the current Bitcoin block for a global clock; per-block rate limits or VDFs can cap message frequency if needed.

4.4.1 Actively Validated Service API

An Actively Validated Service (AVS) interacts with Ember through a domain identifier χ, a payload-submission path, and local decision retrieval. Candidate messages are submitted as raw domain-defined payload bytes for the current ballot in χ; bonded identities wrap those bytes in `ReferendumVote` leaves and emit one payload-bearing attestation per (a_i, χ) chain. The reducer aggregates those endorsements and exposes the resulting `DecisionRoll` or `ImpossibleRoll` through local interfaces such as `getreferendumdecisionroll` and `getdomaininfo`. In the reference node, these flows correspond concretely to RPCs such as `addreferendumvote`, `getreferendumdecisionroll`, and `getdomaininfo`.

Implementation-specific plumbing such as payload-submission RPCs, pending-attestation storage, bond indexing, and network environment glue is summarized in Section 7; it does not change the protocol objects defined above.

4.4.2 Convergence on Endorsement through Leader Selection via Deterministic Sampling

Ember converges through deterministic sampling. For each Zeitgeist (anchored to a Bitcoin block hash) and each domain, a Parliament is sampled and ordered; the *recognized* member is expected to propose the next referendum (or a Decision Roll). Sampling uses fixed public inputs: identity, ballot number, and reference Bitcoin block hash. Because these inputs are deterministic, honest nodes compute the same ranking for the same candidate set.

This uses the Bitcoin block hash [15] as a public seed; miners can grind but at a bounded economic cost. A future VRF beacon [14], commit-reveal beacon [21], or VDF-based signal [3] could further reduce bias without altering the reducer logic.

The core leader-selection algorithm is a batched AVL sum tree (Appendix D) built on the classical AVL balancing scheme [1]. This component is convergence-critical but not strictly consensus-critical: if different software versions compute slightly different leader sets, the system can still progress as long as enough honest stake follows one compatible rule set. In mixed-version periods, performance may degrade (for example, higher latency), but progress can continue when honest-overlap remains high.

4.4.3 Consensus Mechanism

For each domain χ and ballot number b, the reducer applies the following steps:

1. **Collect proposals.** Group observed payload-bearing attestations by candidate value.

2. **Aggregate endorsements.** Measure the active endorsed weight of each candidate at view block B:

$$W^{\chi}_{b,f}(B) = \sum_{s_j \in \mathcal{A}(B)} \omega_j(B)\, \delta\big(s_j \text{ endorses } v^{\chi}_{b,f}\big).$$

3. **Decide or certify impossibility.** If some candidate reaches $W^{\chi}_{b,f}(B) \geq \Theta\Omega(B)$, the node materializes a local `DecisionRoll` for that value. If no candidate reaches quorum, the ballot remains open unless the fixed-view blocking condition of Theorem 3 shows that no candidate can still reach threshold without new endorsements, in which case the node may store an `ImpossibleRoll`. Timeout-driven null-ballots and leader reselection are liveness mechanisms rather than separate endorsement objects; their operational bounds are summarized in Theorem 5 and Section 5.2.

If two candidates tie in endorsed weight, implementations MUST apply a deterministic tie-break rule such as choosing the lowest payload hash. A null-ballot is appropriate only after the leader timeout has elapsed or when the fixed-view blocking test shows that no currently known candidate can still reach quorum without fresh endorsements. The quorum and blocking semantics above are normative; the concrete timeout constants discussed later are operational guidance.

5 Security, Fault Handling, and Game-Theoretic Guarantees

This section proves the core safety and liveness properties used by Ember under the stated assumptions. Economic incentives, operational tuning, and deployment guidance are discussed separately where a full formal treatment is outside the scope of this paper.

5.1 Assumptions and Threat Model

We summarize the core assumptions used throughout the proofs:

(A1) Partial synchrony There exists a time after which the network between honest nodes is synchronous with bounds Δ_{gossip} for envelopes and Δ_{btc} for Bitcoin blocks.

(A2) Honest stake fraction For each Bitcoin block B in the analysis horizon, the honest online stake fraction satisfies $f_{\text{honest}}(B) \geq \Theta$; in particular $f_{\text{honest}}(B) > 1 - \Theta$ whenever safety is invoked.

(A3-up) Upward drift bound For adjacent blocks $B \to B^+$, the active-weight multiplier satisfies $x_\uparrow(B \to B^+) := \Omega(B^+)/\Omega(B) \leq \bar{x}_\uparrow$, with $\bar{x}_\uparrow \geq 1$.

(A3-down) Downward drift bound For adjacent blocks $B \to B^+$, the active-weight multiplier satisfies $x_\downarrow(B \to B^+) := \Omega(B^+)/\Omega(B) \geq \underline{x}_\downarrow$, with $0 < \underline{x}_\downarrow \leq 1$.

(A4) Single-signing Each honest operator signs at most one endorsement per tuple (χ, b, B); across blocks, each honest operator signs at most one value per (χ, b).

(A5) Ballot lock-in Once a node stores a local decision or impossibility certificate for (χ, b) on its canonical chain view, it does not later store a conflicting outcome for the same ballot unless that stored outcome is invalidated by a chain reorganization.

Informally, these correspond to:

- **Stake honesty:** At least two-thirds of active, bonded stake behaves honestly over the time horizon in which leader elections and reductions run (A2, A4).
- **Network:** Bitcoin provides eventual liveness; the Char P2P layer is eventually synchronous with bounded but variable delay (A1).
- **Drift and churn:** Stake growth and decay are tracked separately via $x_\uparrow$ and $x_\downarrow$; downward drift is the critical case for winner flips (A3-up, A3-down).
- **Lock-in behavior:** A finalized ballot is treated as closed in local state unless reorged out (A5).
- **Censorship and cryptography:** Miners may censor some transactions, but we assume at least intermittent inclusion for bond funding and slashing publishes, and standard assumptions on Schnorr signatures and hash preimage/collision resistance.

Economic security. Slash cost is at least the bonded stake plus any burned outputs, so the value at risk must exceed plausible censorship or bribery incentives. The base protocol still lacks a native relay-fee market, so present-day operation depends on participants' interest in finality; the remaining calibration work is deferred to Section 9.

5.2 Formal Guarantees and Bounds

This subsection fixes explicit parameters and states the formal guarantees Char targets. Let:

- Δ_{gossip}: upper bound on P2P attestation propagation among honest nodes during a stable epoch.
- Δ_{btc}: Bitcoin block inter-arrival upper bound used for timeouts (e.g., 20 min).
- Δ_{domain}: A particular domain's maximum ballot fill interval (e.g., 1 min).
- f_{honest}: fraction of active stake that is honest and online in the epoch.
- $x_\uparrow(B \to B') := \Omega(B')/\Omega(B)$ when $\Omega(B') \geq \Omega(B)$, with deployment bound $1 \leq x_\uparrow \leq \bar{x}_\uparrow$.
- $x_\downarrow(B \to B') := \Omega(B')/\Omega(B)$ when $\Omega(B') \leq \Omega(B)$, with deployment bound $\underline{x}_\downarrow \leq x_\downarrow \leq 1$.

- $W_v(B)$: active endorsed weight for value v at block B. When domain/ballot (χ, b) is fixed and $v = v^{\chi}_{b,f}$, this is shorthand for $W^{\chi}_{b,f}(B)$. For fixed endorsement sets (no new signatures), define $m_v(B \to B') := W_v(B')/W_v(B)$.
- k: depth of Bitcoin confirmations an implementation waits before treating a bond funding/roll/slash as stable.
- $\Theta = \frac{2}{3}$: endorsement threshold fraction.

5.2.1 Quorums and Intersection

Definition 1 (Quorum)**.** *For a Bitcoin block B, domain χ, and ballot b, a **quorum** for value v is any set of operators $\mathcal{Q} \subseteq \mathcal{A}(B)$ whose total weight satisfies $\sum_{s_j \in \mathcal{Q}} \omega_j(B) \geq \Theta\Omega(B)$ and whose endorsement sets contain (χ, b, v).*

Definition 2 (Strictly Positive Weight)**.** *We say a quorum (or an intersection of quorums) has **strictly positive honest weight** if the sum of the weights of honest operators in it is greater than zero.*

Lemma 2 (Quorum positivity)**.** *Suppose $f_{honest} > 1 - \Theta$ for block B. Then any quorum $\mathcal{Q}$ contains strictly positive honest weight.*

Proof. Suppose $\mathcal{Q}$ contains no honest weight.

Since $\mathcal{Q}$ is a quorum, it must contain at least $\Theta\Omega(B)$ total weight.

However, honest weight is at least

$$f_{\text{honest}}\Omega(B) > (1 - \Theta)\Omega(B),$$

so dishonest weight is at most

$$(1 - f_{\text{honest}})\Omega(B) < \Theta\Omega(B).$$

This contradicts the assumption that $\mathcal{Q}$ contains no honest weight, as it cannot reach the quorum threshold with only dishonest weight.

Therefore, $\mathcal{Q}$ must contain strictly positive honest weight. □

Lemma 3 (Quorum intersection)**.** *Suppose $f_{honest} > 1 - \Theta$ for block B and let $f_{adversary} = 1 - f_{honest}$ be the fraction of dishonest weight, with $f_{adversary} < \frac{1}{3}$. Equivalently, require $f_{adversary} < 1 - \Theta$; for $\Theta = \frac{2}{3}$, this is $f_{adversary} < \frac{1}{3}$. Let $\mathcal{Q}_1, \mathcal{Q}_2$ be any two quorums for (possibly distinct) values at (χ, b, B). Then $\mathcal{Q}_1 \cap \mathcal{Q}_2$ contains strictly positive honest weight.*

Proof. For any subset $X \subseteq \mathcal{A}(B)$, write $\Omega_B(X) := \sum_{s_j \in X} \omega_j(B)$.

By Lemma 2, both $\mathcal{Q}_1$ and $\mathcal{Q}_2$ contain strictly positive honest weight.

Since honest operators sign at most one value per (χ, b, B) by (A4), the honest weight in $\mathcal{Q}_1$ and $\mathcal{Q}_2$ must be disjoint unless the quorums intersect in some honest operator.

We will prove that $\mathcal{Q}_1$ and $\mathcal{Q}_2$ must intersect by contradiction. First, assume $\mathcal{Q}_1$ and $\mathcal{Q}_2$ are disjoint. Then

$$\Omega_B(\mathcal{Q}_1 \cup \mathcal{Q}_2) = \Omega_B(\mathcal{Q}_1) + \Omega_B(\mathcal{Q}_2) \geq 2\Theta\,\Omega(B).$$

For $\Theta = \frac{2}{3}$, this is $\frac{4}{3}\Omega(B)$, which exceeds the total active weight $\Omega(B)$, a contradiction. Hence $\mathcal{Q}_1$ and $\mathcal{Q}_2$ are not disjoint: $\mathcal{Q}_1 \cap \mathcal{Q}_2 \neq \emptyset$.

Now suppose, for contradiction, that $\mathcal{Q}_1 \cap \mathcal{Q}_2$ contains no honest weight. Then every operator in the intersection is adversarial, so

$$\Omega_B(\mathcal{Q}_1 \cap \mathcal{Q}_2) \leq f_{\text{adversary}}\,\Omega(B).$$

On the other hand, since $\mathcal{Q}_1, \mathcal{Q}_2 \subseteq \mathcal{A}(B)$,

$$\Omega_B(\mathcal{Q}_1 \cup \mathcal{Q}_2) \leq \Omega(B),$$

and by the inclusion–exclusion identity,

$$\begin{aligned}\Omega_B(\mathcal{Q}_1 \cup \mathcal{Q}_2) &= \Omega_B(\mathcal{Q}_1) + \Omega_B(\mathcal{Q}_2) - \Omega_B(\mathcal{Q}_1 \cap \mathcal{Q}_2) \\ &\geq 2\Theta\,\Omega(B) - \Omega(B) \\ &= (2\Theta - 1)\,\Omega(B).\end{aligned}$$

Combining the two bounds gives:

$$f_{\text{adversary}}\,\Omega(B) \;\geq\; \Omega_B(\mathcal{Q}_1 \cap \mathcal{Q}_2) \;\geq\; (2\Theta - 1)\,\Omega(B),$$

so $f_{\text{adversary}} \geq 2\Theta - 1$. For $\Theta = \frac{2}{3}$, this implies $f_{\text{adversary}} \geq \frac{1}{3}$, contradicting the assumption $f_{\text{adversary}} < \frac{1}{3}$.

Therefore, $\mathcal{Q}_1 \cap \mathcal{Q}_2$ must contain strictly positive honest weight. □

5.2.2 Safety and Liveness

Theorem 2 (Safety)**.** *Fix a Bitcoin block B and domain/ballot (χ, b). Under Assumptions (A2) and (A4), and with finalization requiring a quorum of weight at least $\Theta\Omega(B)$ as in Definition 1, at most one value can finalize for (χ, b, B).*

Proof. Assume by contradiction that two distinct values $v \neq v'$ both finalize for (χ, b, B). Let $\mathcal{Q}_v$ and $\mathcal{Q}_{v'}$ be their respective quorums. By Lemma 3, the intersection $\mathcal{Q}_v \cap \mathcal{Q}_{v'}$ contains positive honest weight. Honest operators sign at most one value per (χ, b, B), so no honest stake can belong to both quorums for distinct values. This is a contradiction, hence at most one value can finalize. □

Lemma 4 (Drift ratio dynamics)**.** *Fix (χ, b) and assume no new endorsements are added between blocks B and B'. For values with $W_v(B) > 0$, define:*

$$x := \frac{\Omega(B')}{\Omega(B)}, \qquad m_v := \frac{W_v(B')}{W_v(B)},$$
$$r_v(B) := \frac{W_v(B)}{\Omega(B)}.$$

For values with $W_v(B) = 0$, no-new-endorsement implies $W_v(B') = 0$, hence $r_v(B') = r_v(B) = 0$. For each value v with $W_v(B) > 0$,

$$r_v(B') = \frac{m_v}{x}\, r_v(B).$$

Proof. Substitute definitions:

$$r_v(B') = \frac{W_v(B')}{\Omega(B')} = \frac{m_v W_v(B)}{x\Omega(B)} = \frac{m_v}{x}\, r_v(B).$$

□

Lemma 5 (Drift-up monotonicity criterion)**.** *Under Lemma 4, if $x \geq 1$ and $m_v \leq x$ for all values v with $W_v(B) > 0$, then $r_v(B') \leq r_v(B)$ for all values v. In particular, a value below quorum at B cannot newly cross quorum at B' without new endorsements.*

Proof. For $W_v(B) > 0$, this is immediate from $r_v(B') = (m_v/x) r_v(B)$ with $m_v/x \leq 1$. For $W_v(B) = 0$, the no-new-endorsement condition gives $W_v(B') = 0$, so $r_v(B') = r_v(B) = 0$. □

Lemma 6 (Drift-down winner flip without new signatures)**.** *There exists a fixed endorsement set (no new signatures) and blocks $B_1 < B_2$ such that one value reaches quorum at B_1 while a distinct value reaches quorum at B_2.*

Proof. Let $\Theta = \frac{2}{3}$ and active endorsement weights evolve as:

$$\text{at } B_0: \quad (A, B, C) = (7, 3, 7), \quad \Omega(B_0) = 17,$$
$$\Theta\Omega(B_0) = 11.33\ldots$$

so no value has quorum. After C-weight expires:

$$\text{at } B_1: \quad (A, B, C) = (7, 3, 0), \quad \Omega(B_1) = 10,$$
$$\Theta\Omega(B_1) = 6.66\ldots$$

so A has quorum. Later, most A-weight expires while B-weight remains:

$$\text{at } B_2: \quad (A, B, C) = (1, 3, 0), \quad \Omega(B_2) = 4,$$
$$\Theta\Omega(B_2) = 2.66\ldots$$

so B has quorum. No new endorsements were added between B_1 and B_2. Therefore drift-down can flip the quorum winner without new signatures. □

Lemma 7 (Quorum impossibility from blocking weight)**.** *Fix (χ, b, B) with total active weight $\Omega(B)$, threshold $\Theta \in (0, 1)$, and candidate values V. Let $W_v(B) \geq 0$ be endorsed weight for value v, and assume candidate endorsements are bond-disjoint so $W_{\text{lock}} := \sum_{v \in V} W_v(B) \leq \Omega(B)$. Define $W_{\max} := \max_{v \in V} W_v(B)$. If*

$$W_{\text{lock}} - W_{\max} > (1 - \Theta)\Omega(B),$$

then for every $v \in V$,

$$W_v(B) + \big(\Omega(B) - W_{\text{lock}}\big) < \Theta\Omega(B).$$

Hence no candidate can reach quorum in that fixed view without new endorsements.

Proof. For any v, $W_v(B) \leq W_{\max}$. Let $U := \Omega(B) - W_{\text{lock}}$ be unendorsed active weight. The best case for any candidate is to add all U, giving

$$\begin{aligned} W_v(B) + U &\leq W_{\max} + U \\ &= \Omega(B) - \big(W_{\text{lock}} - W_{\max}\big) \\ &< \Omega(B) - (1 - \Theta)\Omega(B) \\ &= \Theta\Omega(B). \end{aligned}$$

□

Theorem 3 (ImpossibleRoll soundness in a fixed block view)**.** *Fix (χ, b, B) and assume no new endorsements are added. If:*

1. *no candidate value currently satisfies $W_v(B) \geq \Theta\Omega(B)$;*
2. *at least two candidate values have positive endorsed weight;*
3. *$W_{\text{lock}} - W_{\max} > (1 - \Theta)\Omega(B)$;*

then no candidate can reach quorum in that same fixed view. Therefore an `ImpossibleRoll` *is a sound negative certificate for that view.*

Proof. Condition (3) and Lemma 7 imply no candidate can reach $\Theta\Omega(B)$ without new endorsements. Condition (1) rules out already-finalized candidates in this view, and (2) excludes trivial single-candidate cases. Hence a negative certificate is sound for the fixed block context. □

Theorem 4 (Cross-block safety with ballot lock--in)**.** *Under Assumption (A5), a single node on one canonical chain view does not store conflicting finalized outcomes for the same (χ, b) across blocks.*

This theorem is implementation-level and relies on local ballot lock-in semantics.

Proof. By (A5), after a node stores a `DecisionRoll` or `ImpossibleRoll` for (χ, b), later processing does not store a conflicting outcome unless the prior stored outcome is reorg-invalidated. Hence conflicting finalized outcomes are excluded on a fixed canonical chain view. □

Lemma 6 shows that drift-down alone can create winner flips without equivocation. Therefore cross-block safety requires either lock-in semantics (A5) or stronger value-specific drift constraints beyond a single scalar churn bound.

Theorem 5 (Liveness)**.** *Assume (A1) and (A2) hold over the interval considered, and that for each addressed ballot and domain the designated leader (or, after timeout, some honest node) emits at least one proposal or null-ballot within Δ_{domain} of the ballot becoming eligible, with local processing time negligible compared to $\Delta_{gossip}, \Delta_{domain}$. Then for every addressed ballot, eventually either a value or a null-ballot is finalized with a quorum of weight at least $\Theta\Omega(B)$. In the stable synchronous regime, this occurs within $O(\Delta_{domain}+\Delta_{gossip})$ wall-clock time, unless the number of simultaneously active domains exceeds processing capacity.*

This claim depends on partial synchrony [8], online stake, and leader/timeout behavior.

Lemma 8 (Eventual proposal or null-proposal)**.** *Under the assumptions of Theorem 5, for every addressed ballot (χ, b) there exists a time t_0 after the onset of partial synchrony at which some leader or honest node emits a proposal or null-ballot for (χ, b), and all honest nodes receive that envelope by time $t_0 + \Delta_{gossip}$.*

Proof. Fix a ballot (χ, b) and let t^* be a time after which the partial synchrony bounds on Δ_{gossip} and Δ_{btc} hold. By assumption (3) of Theorem 5, once (χ, b) becomes eligible there is a bounded leader or timeout schedule: either the designated leader emits a proposal within Δ_{domain}, or, if it fails to do so, honest nodes detect the timeout and emit a null-ballot within another Δ_{domain}. In either case there exists a finite time $t_0 \geq t^*$ at which some honest process has emitted either a proposal or null-ballot for (χ, b). By partial synchrony, this envelope reaches every honest node within an additional Δ_{gossip}, as required. □

Proof. Fix an addressed ballot (χ, b) and a time after which the partial synchrony assumption holds. By Lemma 8 there is a time t_0 at which some proposal or null-ballot for (χ, b) has been emitted and, by time $t_0 + \Delta_{\text{gossip}}$, received by all honest nodes. Honest nodes either directly endorse the proposal (if valid and timely) or, if a leader is stalled relative to the timeout rule, issue a null-ballot. In either case, each honest node contributes at most one endorsement for this ballot. Since honest stake is at least $\Theta\Omega(B)$ by assumption (1), the union of honest endorsements forms a quorum. Once a quorum is observed, the reducer materializes a Decision Roll and finalizes either the proposed value or the null outcome. The total time from eligibility to finalization in the stable regime is thus bounded by the emission window Δ_{domain} (to reach some t_0) plus one gossip window Δ_{gossip}, yielding the stated latency bound. If at some point honest stake online drops below Θ, no quorum is possible; implementations SHOULD then pause finality and gossip only until the honest fraction recovers, at which point the above argument applies. □

5.3 Relativistic Consensus (Summary)

Finality is defined per Bitcoin block B; attestations bind to the block hash and thresholds use $\Omega(B)$. Define the active set $\mathcal{A}(B) = \{s_i \mid \text{birth}_i + \text{activation\_delay} \leq h < \text{expiry}_i - \text{shutdown\_window}, s_i \text{ not broken}\}$ for block height

h, with $\Omega(B) = \sum_{s_i \in \mathcal{A}(B)} \omega_i(B)$. The formal safety/liveness bounds and drift limits in Section 5.2 apply; they ensure that overlapping honest stake prevents two values from finalizing for the same (χ, b) and that progress occurs when honest online stake exceeds Θ. Reorgs recompute $\mathcal{A}(B)$ and thresholds; attestations for B cannot be replayed to B' because they commit to the block hash.

5.4 Drift, Churn, and Split Finality

Because the active set $\mathcal{A}(B)$ is defined per block, rapid stake churn or long-offline bonds can change thresholds faster than attestations propagate. Two failure modes matter:

- **Stake accretion drift.** New bonds that become active between blocks B and B' increase $\Omega(B')$. An endorsement set that was $\geq \frac{2}{3}\Omega(B)$ may fall below $\frac{2}{3}\Omega(B')$, so a value that appeared final relative to B might not finalize relative to B'.
- **Stake decay/offline drift.** Bonds that expire, quick-break, or remain offline shrink the effective signing power. If honest, online stake drops below $\frac{2}{3}\Omega(B)$, two disjoint minorities could each gather $\frac{2}{3}$ of the diminished live stake and finalize conflicting values in different regions of the network before the active-set view converges.

When splits are possible. Split finality can occur if (a) the honest-online stake is below $\frac{2}{3}$ of the configured active set for the block being attested, and (b) network delay is long enough that two endorsement quorums form without overlap. In practice this requires sustained churn/offline rate plus delayed propagation.

Operational assumptions and monitoring. We assume:

- **Slow churn:** Activation and shutdown windows (six blocks to activate, twelve before expiry) limit instantaneous swings in $\mathcal{A}(B)$.
- **Online fraction:** At least $\frac{2}{3}$ of $\mathcal{A}(B)$ remains online and signing.
- **Timely gossip:** Envelope propagation completes well within the activation/shutdown windows.

Nodes can monitor:

- Measured online weight versus $\Omega(B)$: if observed endorsements plateau below $\frac{2}{3}\Omega(B)$, halt finality for that ballot number and raise an alert.
- Churn velocity: track how much weight enters or leaves $\mathcal{A}(B)$ per block; flag spikes that threaten overlap of endorsement sets.
- Divergent decisions: detect conflicting finalized values per (χ, b, B); treat any occurrence as a critical fault indicating violated assumptions.

Mitigations include lengthening activation/shutdown windows, refusing to finalize when online weight is too low, and rate-limiting new bond activation to cap drift per block.

Lemma 9 (Equivocation detectability)**.** *Consider identity s_i (with anchor id a_i) and its attestation chain as in Section 4.2. Suppose there exist two distinct valid envelopes $\mu_i \neq \mu_i'$ for the same (χ, b). Then there exists a finite witness that any honest node can verify which proves that s_i equivocated and from which s_i's signing key can be extracted.*

The proof sketch below depends only on message availability for a direct same-topic witness. For payload-bearing histories with the same repeated genesis binding, this is a same-$(\chi, b, G_{i,\chi})$ pair. If two observed histories instead carry different signed genesis bindings, the conflict is already pinned to ballot 0, i.e. a zero-walk case. A backward walk is needed only to localize an ancestry-consistency violation against the repeated genesis binding.

Proof sketch. By construction, every valid attestation for domain χ and ballot b carries a Purify proof for the same topic $T(\chi, b)$. Verification recovers a unique public nonce $R_{\chi,b}$ from that proof and requires it to equal the Schnorr signature's nonce point. Therefore any two distinct valid attestations for the same (χ, b) necessarily reuse the same nonce. Since the messages differ, their Schnorr challenges differ, and the standard nonce-reuse attack recovers the signing key (Appendix B). For payload-bearing histories that share the same repeated genesis binding $G_{i,\chi}$, the witness is just the pair of conflicting attestations plus the bond's public key material. The same argument applies to genesis itself, using

topic $T(\chi, 0)$: if two observed histories carry different signed genesis bindings, then the two genesis attestations already form the direct ballot-0, or zero-walk, witness. Thus any direct same-topic witness is position-independent once the conflicting topic is identified. A backward walk is required only when one is trying to localize a later-discovered ancestry violation, such as a descendant chain whose first payload-bearing ancestor contradicts its declared $b^{\chi}_{\text{start},i}$. □

In particular, if conflicting values ever do finalize for the same (χ, b) within a window where the corresponding active sets still overlap in at least $(1 - \Theta)$ honest weight, then some of that overlapping stake must have signed twice and is slashable, by Lemma 9. If, on the other hand, the overlap assumption itself is violated (for example due to excessive churn or effective network replacement), then safety can fail without equivocation; such a failure manifests as a breach of the drift/overlap operating assumptions monitored in Section 5.4.

Safety (cross-block drift). For a fixed block B, single-view safety is Theorem 2. Across blocks, Lemma 5 gives the safe upward-drift criterion, while Lemma 6 shows downward drift can change winners with no new signatures. Theorem 3 analogously scopes negative-finality claims to a fixed block view.

Accordingly, the reference implementation achieves cross-block safety for a ballot primarily by ballot lock-in (A5): once a roll is stored, that ballot is locally closed unless reorg-invalidated.

Drift policy bounds. Use separate growth/decay multipliers:

$$1 \le x_{\uparrow} \le \bar{x}_{\uparrow}, \qquad \underline{x}_{\downarrow} \le x_{\downarrow} \le 1.$$

In addition to tracking $\Omega(B)$, operators SHOULD monitor value-specific drift ratios m_v/x for values with $W_v(B) > 0$. If a previously non-leading value's m_v/x rises above 1 while a ballot is still open, implementations SHOULD pause that ballot until canonical context converges.

Reorg tolerance. Reorg tolerance relies on blockhash-contextual rolls plus ballot lock-in.

Bonds are only activated after 6 blocks and shut down 12 blocks before expiry; implementations SHOULD require $k \ge 6$ confirmations on bond/roll/slash publishes.

If a reorg invalidates the block context for a stored roll, that roll can be reopened and recomputed under the new canonical context. Deep reorgs can therefore still create temporary divergence across nodes if they lock at different times; operators SHOULD monitor reorg depth and pause processing when convergence assumptions are not met.

Liveness (per block, per domain, per ballot). This is the operational form of Theorem 5. Assume:

1. $f_{\text{honest}} \ge \Theta = \frac{2}{3}$ remains online.
2. Network delivery among honest nodes within Δ_{gossip}.
3. Leaders (or null-fillings) emit one proposal per domain within Δ_{domain}.

Then every addressed ballot obtains a value endorsed by at least $\Theta\Omega(B)$ within $O(\Delta_{\text{gossip}} + \Delta_{\text{domain}})$ wall-clock time, unless the number of simultaneously active domains exceeds processing capacity.

Timeout calibration. The following is recommended operational tuning rather than a proved bound in this paper. Let $T_{\text{leader}} = \Delta_{\text{gossip}} + \delta$ for small slack δ; if no leader proposal arrives by T_{leader}, nodes MAY emit a null-ballot. Mutiny downgrades SHOULD wait at least one additional Δ_{gossip} to ensure leaders are actually stalled.

5.5 Operational Assumptions and Monitoring

Parameter	Recommended stance
Online stake	SHOULD target $f_{\text{honest}} \geq \Theta = \frac{2}{3}$ of $\mathcal{A}(B)$; alarm if observed online weight drops below $\Theta\Omega(B)$.
Upward drift	SHOULD keep $x_{\uparrow}$ close to 1 per block (e.g., $x_{\uparrow} \leq 1.05$); rate-limit activations when exceeded.
Downward drift	SHOULD keep $x_{\downarrow}$ close to 1 per block (e.g., $x_{\downarrow} \geq 0.95$); pause open ballots when rapid expiry/offline decay is observed.
Gossip budget	SHOULD keep attestation propagation within Δ_{gossip}; raise null-ballot after T_{leader}.
Reorg depth	MUST wait $k \geq 6$ confs for bond/roll/slash stability; pause finality on deeper reorgs.
Payload cap	MUST enforce 3 MB Bamboo payload per attestation.

6 Parameters and Expected Performance

The latency and throughput figures in this section are operational estimates under the assumptions above.

Parameter	Example stance
Bitcoin block cadence	10 min target; Zeitgeist anchored per tip.
Gossip delay Δ_{gossip}	2–5 s across char-relay peers (tunable by deployment).
Domain pacing Δ_{domain}	1 s for high-frequency AVSes; coarser for low-rate feeds.
Payload cap	3 MB Bamboo payload per attestation.
Active bond scale	Thousands of bonds sampled; sampler runs in $O(\log N)$.

With the above knobs, common-case Decision Rolls emerge in $\Delta_{\text{domain}} + \Delta_{\text{gossip}} \approx 3{-}6\,\text{s}$ from payload eligibility, assuming leaders emit promptly and online stake $\geq \Theta$. Worst-case null-ballot fallback adds one more Δ_{gossip}.

Aggregate data rate per AVS is roughly:

$$\text{throughput}_{\chi} \approx \frac{N_{\text{online},\chi} \cdot S_{\chi}}{T_{\text{btc}}},$$

where $N_{\text{online},\chi}$ is online signing identities for domain χ, $S_{\chi} \leq 3$ MB is average per-attestation Bamboo payload carried for that domain, and T_{btc} is the Bitcoin block interval. Domain traffic scales by emitted domain-chain attestations (one domain payload per attestation), plus occasional signed genesis attestations when a bond first opens a domain chain.

Deployment guidance. Operators SHOULD measure observed Δ_{gossip} and adjust Δ_{domain} and timeouts accordingly; if online stake drops below Θ or drift exceeds targets, nodes SHOULD pause finality and continue gossip-only until overlap recovers.

7 Reference Implementation Snapshot

The reference implementation is organized around four components: bond onboarding and indexing, attestation production, local tabulation, and network/-tooling surfaces. Bond-creation RPCs build anchors and stake outputs, while `charbondindex` tracks active bonds and feeds the weighted sampler. The attestation worker batches pending payloads by domain, ensures genesis exists, adds any needed reorg headers, and signs one payload-bearing attestation per domain with the anchor key plus a Purify proof [9]. `ReferendumTabulationDB` records endorsements and stores locally derived `DecisionRoll` and `ImpossibleRoll` objects. The network layer gossips envelopes over `charenv`, and the RPC surface covers payload submission, attestation inspection, and decision retrieval.

The main extensions still outside the present codebase are covenant-grade or miner-independent slashing, light-client or ZKP compression for Bamboo datasets [20, 12, 19], and richer timeout or mutiny recovery beyond deterministic leader reselection.

8 Practical Implementation Details and Node Operations

8.1 Worked Example

Consider ballot (χ, b) with two candidate payloads, "A" and "B", under a fixed Bitcoin/stake view B. Suppose operators s_1, s_2, s_3 emit valid `ReferendumVote` attestations, where s_1 and s_2 endorse "A" and s_3 endorses "B". If $\omega_1(B)+\omega_2(B) \geq \Theta\Omega(B)$, then the node groups those attestations by payload, observes that "A" reaches quorum, and materializes a local `DecisionRoll` for "A" with `CheckableAttestation` proofs for s_1 and s_2. If later evidence shows that one of those signers equivocated on the same topic, Purify nonce reuse yields the signing key, the bond can be quick-broken or slashed, and any still-open ballots that depended on that signer's weight are recomputed under the reduced active set.

8.2 Handling Slashed and Equivocating Bonds

Intuition for slashing. Purify binds every valid attestation for (χ, b) to the same topic-specific nonce, so signing two conflicting messages for that same topic reuses the nonce and exposes the private key. That makes equivocation fatal rather than merely blameworthy: the protocol can move directly from conflicting signatures to key extraction and then to on-chain accountability.

Lemma 9 is the normative equivocation statement. Operationally, once a node has either a direct same-$(\chi, b, G_{i,\chi})$ witness or the corresponding ballot-0 zero-walk witness from conflicting signed genesis attestations, it can recover the signing key, publish quick-break or slash transactions, and mark the bond as tombstoned to aid convergence.

Let B denote the Bitcoin view block used for post-slash recomputation. The reducer then removes s_i's weight from the active set,

$$\Omega_{\text{new}}(B) = \Omega(B) - \omega_i(B), \qquad s_i \notin \mathcal{A}(B),$$

and recomputes any still-open ballots under the same reduction rules as before, but without s_i's endorsements. This post-slash handling improves accountability and can change outcomes for ballots that were still open, but it does not replace Theorems 2 and 4; ballots already locked in under (A5) remain governed by lock-in semantics unless their Bitcoin context is reorg-invalidated. AVSs that execute ballot outcomes should therefore treat slash-driven recomputation like any other rollback-and-replay event over the still-open suffix of the ballot stream.

8.3 Long-Range Attacks and Anchor/Chain Tip Commitments

This subsection provides threat-model reasoning and mitigation design constraints rather than a full theorem/proof pair.

Long-range attacks in proof-of-stake style systems typically arise [11, 4, 5, 6] when old signing keys or stake positions are repurposed to fabricate plausible but fraudulent histories far in the past. Char constrains this attack surface in two complementary ways: by making bonds explicitly time-bounded and by having anchor-scoped identities commit to their own attestation tips on exit.

Per-anchor tip commitments on timeout Consider identity s_i (anchor id a_i) with relative timeout τ, funded at Bitcoin block B_{open} and scheduled to become spendable at $B_{\text{close}} = B_{\text{open}} + \tau$. In the reference design, a "natural" shutdown tied to anchor a_i (the CSV-path exit as opposed to a slash or roll) is extended with a commitment to the tip set of s_i's attestation namespace:

- Let H_i^{tip} be a commitment to the latest envelope tip(s) that s_i considers valid for its own history at or before B_{close}.[4]
- The shutdown transaction for a_i includes H_i^{tip} in an `OP_RETURN` or script witness, so that any later verifier can check that s_i's published attestation chain is a prefix of a chain rooted at H_i^{tip}.

Once such a shutdown has been mined and buried under k Bitcoin confirmations, any alternative history in which s_i participates must either:

1. extend the same envelope tip H_i^{tip}, or
2. contradict the on-chain commitment attached to the shutdown of a_i.

[4] A compact Merkle-aggregated vector [13] over active domain-chain tips is one natural representation.

Honest nodes and AVSes can adopt an objective rule of ignoring histories where an identity's claimed envelopes are not consistent with its on-chain tip commitments, thereby eliminating many long-range rewrite attempts that rely on "forgetting" earlier messages. Combined with predecessor-linked attestation histories and locally derived roll proofs (Section 4.2), these per-anchor/per-chain tip commitments ensure that any equivocation that ever contributed weight to a `DecisionRoll` must appear as an inconsistency in the graph of attestations and can be detected without trusting a separate, centralized checkpoint stream.

Comparison to centralized checkpointing schemes Many PoS and sidechain systems rely on "central" checkpointing schemes: a foundation-run key, a hard-coded validator committee, or a small set of operators periodically publishes the canonical state root. Clients must either trust those keys or rely on social consensus to decide which checkpoint stream to follow.

The Char design is more bottom-up and local:

- Bitcoin provides an objective, censorship-resistant log onto which per-anchor/per-chain tip commitments are anchored.
- No single committee can unilaterally roll back history; any proposed alternate view must be consistent with all previously mined bond exits, which are keyed to individual stake positions.
- The trust assumption collapses to the usual stake honesty assumption (weighted majority of stake behaves) plus Bitcoin's PoW security, rather than the honesty of a special-purpose checkpoint signer.

This combination narrows the long-range attack surface while avoiding any new central point of control over finality.

Relation to prior long-range defenses The proof-of-stake and BFT literature offers several additional long-range mitigations that Char does not adopt in its base design [7, 22, 11, 5, 6]:

- **Weak subjectivity checkpoints.** Systems such as early Ethereum PoS require clients to accept a periodically published "weak subjectivity" checkpoint from a social or protocol authority [4, 5, 6]. Char omits this mechanism because it reintroduces a notion of special checkpoint signers; instead, the only hard commitments originate from ordinary bonds exiting with H_i^{tip} and the attestation histories and roll proofs already needed for normal verification.
- **Inactivity leaks and stake decay.** Some PoS protocols gradually reduce the effective weight of non-participating validators to make long-range attacks by dormant keys harder [6]. Char keeps stake weights simple and objective: a bond either exists on-chain with a given amount or it does not. This avoids embedding time-varying, implementation-dependent penalties into the consensus reducer.
- **Key-evolving signatures.** Key-evolving or epoch-based keys can prevent an attacker from signing convincing ancient histories [2], but they also complicate initial synchronization and auditing: a verifier must track which key version was valid at which time in order to decide whether an old signature is acceptable. Char instead keeps keys static over a bond's lifetime and derives protection from the bond's explicit time bounds and its on-chain tip commitment H_i^{tip}, so that initial synchronization only has to reason about which bonds were live when, rather than about implicit key epochs.
- **Global bootstrap rules.** Many designs hard-code client bootstrap rules such as "refuse to sync from views older than T" or "require a committee-signed checkpoint before height h" [4, 6]. Char leaves such bootstrap policies to implementations and operators, while the protocol itself provides only objective artifacts (bond exits, attestation histories, `DecisionRoll`s) from which those policies can be defined.

In summary, the long-range defence strategy in Char is deliberately minimal. It relies on bond time-boundedness, per-anchor/per-chain tip commitments, and attestation/roll evidence to make equivocation objectively detectable, and leaves higher-layer convenience defences such as social

checkpoints or stake-decay heuristics to optional deployment practice rather than to the core protocol.

9 Future Work

- Complete machine-checked proofs of safety, liveness, and long-range resistance under the stated model.
- Tighten timeout/backoff parameterization and provide deployment-grade defaults backed by field measurements.
- Specify stronger anti-censorship relay incentives and recovery workflows for prolonged network partitions.
- Finalize operator playbooks for churn, emergency key rollover, and adversarial reorg handling.

References

[1] Georgy M. Adelson-Velsky and Evgenii M. Landis. An algorithm for the organization of information. *Proceedings of the USSR Academy of Sciences*, 146:263–266, 1962.

[2] Mihir Bellare and Sara K. Miner. A forward-secure digital signature scheme. In *Advances in Cryptology – CRYPTO '99*, volume 1666 of *Lecture Notes in Computer Science*, pages 431–448, 1999.

[3] Dan Boneh, Joseph Bonneau, Benedikt Bünz, and Ben Fisch. Verifiable delay functions. In *Advances in Cryptology – EUROCRYPT 2018*, volume 10820 of *Lecture Notes in Computer Science*, pages 757–788, 2018.

[4] Vitalik Buterin. Proof of stake: How i learned to love weak subjectivity, November 2014. Ethereum Foundation blog post.

[5] Vitalik Buterin and Virgil Griffith. Casper the friendly finality gadget. `https://arxiv.org/abs/1710.09437`, 2017. CoRR abs/1710.09437.

[6] Vitalik Buterin, Diego Hernandez, Thor Kamphefner, Khiem Pham, Zhi Qiao, Danny Ryan, Juhyeok Sin, Ying Wang, and Yan X. Zhang. Combining GHOST and casper. `https://arxiv.org/abs/2003.03052`, 2020. CoRR abs/2003.03052.

[7] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation (OSDI '99)*, pages 173–186. USENIX Association, 1999.

[8] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2):288–323, 1988.

[9] Judica, Inc. Purify. `https://judica.org/purify/`, 2026. Implementation landing page.

[10] Judica, Inc. Purify and zero-knowledge proofs in `purify-cpp`. `https://judica.org/purify/paper/purify-paper.pdf`, March 2026. Technical note.

[11] Aggelos Kiayias, Alexander Russell, Bernardo David, and Roman Oliynykov. Ouroboros: A provably secure proof-of-stake blockchain protocol. In *Advances in Cryptology – CRYPTO 2017*, volume 10401 of *Lecture Notes in Computer Science*, pages 357–388, 2017.

[12] Robin Linus. BitVM: Compute anything on bitcoin. `https://bitvm.org/bitvm.pdf`, 2023. White paper.

[13] Ralph C. Merkle. A certified digital signature. In *Advances in Cryptology – CRYPTO '89 Proceedings*, volume 435 of *Lecture Notes in Computer Science*, pages 218–238, 1989.

[14] Silvio Micali, Michael O. Rabin, and Salil P. Vadhan. Verifiable random functions. In *40th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 120–130, 1999.

[15] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. `https://bitcoin.org/bitcoin.pdf`, 2008. White paper.

[16] Jonas Nick, Tim Ruffing, Yannick Seurin, and Pieter Wuille. MuSig-DN: Schnorr multi-signatures with verifiably deterministic nonces. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 1717–1731, 2020.

[17] J. Nick P. Wuille and T. Ruffing. BIP 340: Schnorr signatures for `secp256k1`. `https://github.com/bitcoin/bips/blob/master/bip-0340.mediawiki`, 2020. Bitcoin Improvement Proposal.

[18] J. Nick P. Wuille and AJ Towns. BIP 341: Taproot—segwit version 1 spending rules. `https://github.com/bitcoin/bips/blob/master/bip-0341.mediawiki`, 2020. Bitcoin Improvement Proposal.

[19] Jeremy Rubin. Delbrag: A kawaii garbled circuit for bitvm. `https://rubin.io/public/pdfs/delbrag.pdf`, 2025.

[20] Jeremy Rubin. Un-fed covenants: Char-ting a new path to emulated covenants via bitvm integrity checks. `https://rubin.io/public/pdfs/unfedcovenants.pdf`, 2025.

[21] Andrew C. Yao. How to generate and exchange secrets. In *27th IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 162–167, 1986.

[22] Maofan Yin, Dahlia Malkhi, Michael K. Reiter, Guy Golan-Gueta, and Ittai Abraham. Hotstuff: BFT consensus with linearity and responsiveness. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, pages 347–356, 2019.

Appendices

A Bond Contract Example

The following Rust-like contract fragment demonstrates the core logic of an attestation chain.

The contract defines states (e.g., `Operational`, `Closing`) and enforces rules for transitioning between them, handling slashing upon detection of equivocation. The `cheated` function specifies the penalty by burning the funds through an `OP_RETURN` output, ensuring that neither the bond controller nor any colluding party can reclaim them.

```
// Staking States (Operational, Closing)

// Operational State
// State where stakes should be recognized for voting
pub struct Operational;
// Closing State
// State where stakes are closing and waiting evidence of misbehavior
struct Closing;
// enum trait for states
pub trait StakingState {}
impl StakingState for Operational {}
impl StakingState for Closing {}

// Staker is a contract that proceeds from Operational -> Closing
// During it's lifetime, many things can be signed with signing_key,
// but should the key ever leak (e.g., via nonce reuse) the bonded
// funds can be burned.
//
// Burning is important v.s. miner fee because otherwise the staker
// can bribe (or be a miner themselves) to cheat.
pub struct Staker<T: StakingState> {
    // How long to wait for evidence after closing
    timeout: AnyRelTimeLock,
    // The key that if leaked can burn funds
    signing_key: XOnlyPublicKey,
    // The key that will be used to control & return the redeemed funds
    redeeming_key: XOnlyPublicKey,
    // current contract state.
    state: PhantomData<T>
}
```

```
// Functional Interface for Staking Contracts
pub trait StakerInterface
where
    Self: Sized + Contract,
{
    // The key used to sign messages
    decl_guard!(staking_key);
    // the clause to begin a close process
    decl_guard!(begin_redeem_key);
    // the clause to finish a close process
    decl_guard!(finish_redeem_key);
    // The transition from Operational to Closing
```

```
    decl_then!(begin_redeem);
    #[then(guarded_by=[Self::staking_key])]
    fn cheated(self, ctx: Context) {
        let f = ctx.funds();
        ctx.template()
            .add_output(f, &Compiled::from_op_return(b"burn")?, None)?
            .into()
    }
}

impl StakerInterface for Staker<Operational> {
    // redeeming key
    #[guard]
    fn begin_redeem_key(self, _ctx: Context) {
        Clause::Key(self.redeeming_key)
    }
    // begin redemption process
    #[then(guarded_by = [Self::begin_redeem_key])]
    fn begin_redeem(self, ctx: Context) {
        let f = ctx.funds();
        ctx.template()
            .add_output(
                f,
                &Staker::<Closing> {
                    state: Default::default(),
                    timeout: self.timeout,
                    signing_key: self.signing_key,
                    redeeming_key: self.redeeming_key,
                },
                None,
            )?
            .into()
    }
    // staking key
    #[guard]
    fn staking_key(self, _ctx: Context) {
        Clause::Key(self.signing_key)
    }
}
```

```
impl StakerInterface for Staker<Closing> {
    #[guard]
    fn finish_redeem_key(self, _ctx: Context) {
        Clause::And(vec![Clause::Key(self.redeeming_key), self.timeout.into()])
    }
    #[guard]
    fn staking_key(self, _ctx: Context) {
        Clause::Key(self.signing_key)
    }
}

impl<T: 'static + StakingState> Contract for Staker<T>
where
```

```
    Staker<T>: StakerInterface,
    T: StakingState,
{
    declare! {then, Self::begin_redeem, Self::cheated}
    declare! {finish, Self::finish_redeem_key}
    declare! {non updatable}
}
```

B Leaking Key Through Nonce Reuse

The below code sample is a C implementation of the algorithm to extract a private key (nonce) from two Schnorr signatures using scalar arithmetic on the `secp256k1` curve.

Parameters

`sig1` First signature value (32 bytes).

`sig2` Second signature value (32 bytes).

`msg1` Hash of the first message (32 bytes).

`msg2` Hash of the second message (32 bytes).

`out` Output buffer for the extracted nonce (32 bytes).

Implementation outline

1. Convert the five input byte-arrays to curve scalars, aborting on overflow.
2. Recall the signature equations $s_1 = m_1\,x + k$ and $s_2 = m_2\,x + k$, where x is the private key and k the nonce.
3. Subtract the two equations: $s_1 - s_2 = (m_1 - m_2)\,x$.
4. Compute $\Delta_s = s_1 - s_2$ and $\Delta_m = m_1 - m_2$.
5. Verify $\Delta_m \neq 0$ to avoid division by 0.
6. Compute $x = \Delta_s\,\Delta_m^{-1} \pmod{n}$, where n is the curve order.

Soundness note The above derivation is the standard nonce-reuse attack on Schnorr-style signatures: under the usual assumptions on the curve group and signature encoding, reusing a nonce k in two distinct messages uniquely determines the private key x. Implementations must ensure that the (m_1, m_2, s_1, s_2) values are extracted consistently from the wire format, and handle edge-cases such as alternative encodings (for example, low-s normalisation) to avoid misinterpreting malformed signatures.

Return value `true` if all arithmetic succeeded; `false` otherwise.

```
bool char_crypto_extract_nonce(
 const unsigned char* sig1,
 const unsigned char* sig2,
 const unsigned char* msg1,
 const unsigned char* msg2,
 unsigned char* out)
{
    secp256k1_scalar s1, s2, msg1_scalar, msg2_scalar;
    int overflow;
    secp256k1_scalar_set_b32(&s1, sig1, &overflow);
    if (overflow) {
        return false;
    }
    secp256k1_scalar_set_b32(&s2, sig2, &overflow);
    if (overflow) {
        return false;
    }
    secp256k1_scalar_set_b32(&msg1_scalar, msg1, &overflow);
    if (overflow) {
        return false;
    }
    secp256k1_scalar_set_b32(&msg2_scalar, msg2, &overflow);
    if (overflow) {
        return false;
    }

    // s1 = m1*x + k
    // s2 = m2*x + k
    // s1-s2 = (m1*x + k) - (m2*x + k)
    // s1-s2 = (m1 - m2)*x
    // (s1-s2) (m1 - m2)^-1 = x

    // Calculate s1 - s2
    secp256k1_scalar neg_s2;
    secp256k1_scalar s_diff;
    secp256k1_scalar_negate(&neg_s2, &s2);
    secp256k1_scalar_add(&s_diff, &s1, &neg_s2);

    // Calculate msg1 - msg2
    secp256k1_scalar neg_msg2;
    secp256k1_scalar msg_diff;
    secp256k1_scalar_negate(&neg_msg2, &msg2_scalar);
    secp256k1_scalar_add(&msg_diff, &msg1_scalar, &neg_msg2);

    // Check if msg_diff is zero to avoid division by zero
    if (secp256k1_scalar_is_zero(&msg_diff)) {
        return false;
    }

    // Calculate inverse of msg_diff
    secp256k1_scalar msg_diff_inv;
    secp256k1_scalar_inverse_var(&msg_diff_inv, &msg_diff);

    // Calculate private key: (s1 - s2) * (msg1 - msg2)^-1
    secp256k1_scalar private_key;
    secp256k1_scalar_mul(&private_key, &s_diff, &msg_diff_inv);

    // Convert to bytes
    secp256k1_scalar_get_b32(out, &private_key);
    return true;
}
```

Listing 1: Extracting a private key from two Schnorr signatures

C Example of Equivocation History

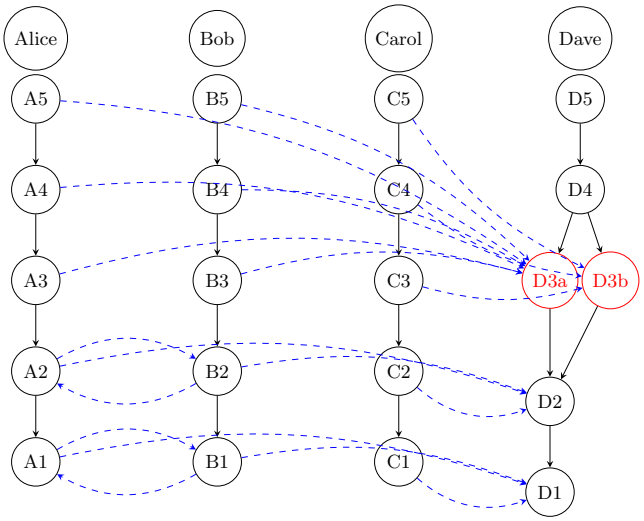

Figure 2: Attestation Chains with Dave Equivocating at Message 3

D AVL Sum Tree with Batching

The AVL Sum Tree is a data structure designed to efficiently manage random sampling from a set of bonded identities. The details are:

- **AVL Sum Tree Structure:** Proposals from bonded identities are organized in a self-balancing AVL tree where each interior node stores the sum of staked endorsements for its subtree. This structure ensures O(log n) complexity for insertions, deletions, and queries.
- **Batched Sampling Mechanism:** Multiple proposals and endorsements are processed as batches. Batching minimizes the overhead of multiple queries to only have to traverse the tree at most once, leading to amortized complexity of $E[O(1)]$ per marginal sample.
- **Determinism:** The AVL-sum tree is deterministic. For a given seed, the sample will always return the same value.
- **Commutativity:** The tree can be built correctly by any order of inserts and deletes; the current membership is all that matters for consistent sampling across nodes. [5]
- **Immutability for Sampling:** The data structure is not mutated at any point for the sampling process. The history of sampling does not affect future samples.
- **Platform Independent:** Only integer math is used; no floating point operations are involved. The code is written carefully to avoid any undefined behavior. Irrespective of platform, the same code will produce the same result on any machine.
- **ZK-Sampling:** Given the emphasis on a fully deterministic tree, the sampling algorithm can be implemented in a ZK-toolchain such as Cairo, and proofs of leader selection for a given set of ballot numbers can be distributed. Since the tree is also buildable deterministically, nodes could only need to store a a commitment of the inserts and deletes in order to be able to verify a proof of a sampling.

[5] In fact, an even stronger property holds. Given two identical trees, if a single perturbation of weight of size P is applied (e.g., deleting a member) to one and not the other, a given bond with weight B's likelihood of being sampled by both is only changed by

$$\min\Big(\frac{P}{B}, 1\Big).$$

This property ensures that the sampling process is robust to small changes in the tree structure, which, if tracked, is also post-hoc detectable. In other words, one can determine if the total number of changes has been large enough to cause this selection to be no longer valid. Multiple updates may cancel out, so the analysis is less straightforward in that context. This enables some amount of "inconsistent filtering" rules to be applied in certain contexts, in particular against small bonds while not impacting which big bonds are sampled.

E Message Ordering and Transitive Inclusion Optimizations

This appendix is non-normative and describes optimization patterns for message discovery and ordering. Safety and liveness do not depend on these optimizations.

E.1 Discovery Hints and Fetch Order

Nodes may accelerate discovery by gossiping inventory for recently observed attestations and, in some implementations, by attaching non-normative cross-links to nearby messages.[6] Such hints can help peers fetch missing predecessors, signed genesis attestations, or neighboring domain-chain messages with lower latency.

Implementations may also choose any deterministic local fetch order when several candidate messages are discovered at once. This affects bandwidth use and time-to-availability, but not protocol validity: admissibility is determined by the attestation's own predecessor commitment, repeated genesis binding, ballot progression, Bitcoin header context, and topic-bound signature proof.

E.2 What Cross-links Do Not Mean

Cross-links or discovery references do not create endorsements, do not change voting weight, and do not induce an alternative ballot ordering rule. In Ember, the voting act is the payload-bearing attestation itself, and the reducer counts endorsements from valid attestations for the addressed (χ, b) topic only.

Consequently, if a node first learns about disagreement through a later tip or an auxiliary reference, it may use those hints to fetch the relevant ancestry more quickly, but the eventual reducer outcome still depends only on the validated attestation set and the local roll derivation rules described in the main body. Discovery order may change latency; it does not change consensus semantics.

[6] These cross-links are an implementation convenience for discovery and repair. They are not part of the minimal Ember validity rules.