Zenon Greenpaper Series

A Verification-First Architecture for Dual-Ledger Systems

Status: Community-authored greenpaper (non-normative, non-official)

Abstract

This paper presents a unified architecture for resource-bounded verification in dual-ledger distributed systems. Unlike traditional blockchains, which treat verification as a byproduct of execution, this design elevates verification to a foundational principle. Execution itself is constrained to remain verifiable under explicitly declared resource limits.

The architecture separates parallel execution from sequential commitment ordering through three tightly integrated pillars:

1. **Bounded Verification** – Verification under explicit resource constraints, anchored to genesis trust roots with adaptive retention.
2. **Proof-Native Applications (zApps)** – Applications where correctness is established via cryptographic proofs rather than execution replay.
3. **Composable External Verification (CEV)** – Trustless validation of external facts (e.g., Bitcoin) without relying on intermediaries.

Operating under strict constraints–$O(N)$ storage for N Momentum block headers, $O(\log m)$ commitment inclusion proof size (Merkle branch) for a commitment under r_C (where m is commitments per Momentum block, per Definition 2.4.1), and browser-native computation–the system enables independent verification even by lightweight clients. Participants can verify correctness without continuous connectivity or global state reconstruction.

This model inverts traditional blockchain priorities: verifcation is foundational; execution is secondary. Each verifer operates within declared resource and trust boundaries and can honestly refuse queries exceeding them. This principle, **refusal as correctness** defnes a new paradigm for distributed systems built for resource-constrained reality.

---

1. Introduction

1.1 The Verification-Execution Tension

In most blockchains, verification equals replay. To verify a transaction, a node must re-execute the computation that produced it. As transaction volume and application complexity increase, this equivalence creates an unavoidable scaling problem: **verifiers must match the resource profile of executors**.

This model excludes lightweight participants–browsers, mobile devices, intermittently connected nodes–from independent validation. Systems optimized for throughput demand trusted intermediaries; systems optimized for verification limit expressiveness. As state grows into hundreds of gigabytes and execution environments evolve, this tension increasingly favors centralized infrastructure.

1.2 From Execution-First to Verification-First

This paper proposes an alternative: a **verification-first architecture**.

Instead of adapting verification to keep pace with execution, we design execution to remain verifiable. The key question becomes:

> *"What forms of execution can remain verifiable under explicit resource bounds?"*

The answer lies in a dual-ledger design separating two concerns:

- **Account-chain layer (execution)**: Each account maintains its own append-only ledger of state transitions, enabling parallel execution without coordination bottlenecks.
- **Momentum chain layer (commitment ordering)**: A global sequential ledger that records cryptographic digests (commitments) of account-chain state transitions, providing temporal ordering and global anchoring.

This separation allows accounts to process transactions independently while maintaining a verifiable global order. Verifiers only track the accounts they care about–anchoring them to the global Momentum chain for trust-minimized synchronization.

Three architectural pillars emerge from this separation:

1. **Bounded Verification (§2)** – Verification with explicit limits on storage, bandwidth, and computation.
2. **Proof-Native Applications (§3)** – Applications where correctness is proven, not replayed.

3. **Composable External Verification (§4)** – Verification of external facts using cryptographic proofs instead of trusted intermediaries.

1.3 Architectural Principles

Four core principles govern the system:

1. **Verification as Foundation**: Execution exists to produce verifiable state transitions. Computations that cannot be efficiently verified are architecturally excluded.
2. **Explicit Resource Bounds**: Every verification operation declares its storage (S), bandwidth (B), and computation (C) bounds upfront. Verifiers refuse queries exceeding these limits–there are no "best effort" fallbacks.
3. **Genesis Anchoring**: Trust roots are embedded at genesis. Any verifier, even after long offline periods, can resynchronize by following cryptographic commitment chains without social coordination.
4. **Honest Refusal**: When a verifier cannot cryptographically prove correctness within its bounds, it refuses instead of trusting. Refusal is explicit, deterministic, and surfaced to users as a correctness guarantee, not a failure.

1.4 What This Architecture Enables

By combining dual-ledger separation, bounded verification, proof-native applications, and external verification, this architecture achieves properties rarely found together:

- **Browser-Native Verification**: Lightweight clients verify state transitions directly via cryptographic proofs.
- **Long-Offline Recovery**: Clients resynchronize by following commitment chains from their last verified header (or an optional locally stored checkpoint)–no social checkpoints required.
- **Cross-Chain Validation**: Bitcoin transactions can be verified trustlessly through SPV-style proofs.
- **Proof-Carried Execution**: Applications execute off-chain and submit proofs verifiable in constant time on constrained devices.

These capabilities allow secure participation even under intermittent connectivity and limited storage.

1.5 What This Architecture Does Not Provide

To preserve bounded, cryptographic verification, several properties are explicitly sacrificed:

- **No Global Atomic Transactions**: Cross-account operations are asynchronous; atomicity across arbitrary accounts is not guaranteed.
- **No Unbounded Compositional Verification**: Verifiers declare finite scope (e.g., "last 1000 commitments") and refuse queries beyond it.
- **No Universal Liveness Under Partition**: Offline verifiers cannot validate new commitments until reconnection.
- **No Censorship Resistance for Proof Distribution**: While commitment ordering is censorship-resistant, proof distribution depends on off-chain networks that may selectively withhold data.

These are not flaws but formalized trade-offs. They define the precise boundaries of verifiability under constrained resources.

1.6 Roadmap

The remainder of this paper proceeds as follows:

- **Section 2**: Formalizes bounded verification–defining commitment chains, trust roots, predicates, refusal semantics, and retention policies.
- **Section 3**: Introduces proof-native applications (zApps), where correctness is cryptographically attested.
- **Section 4**: Extends verification to external systems (notably Bitcoin) through composable external verification (CEV).
- **Section 5**: Demonstrates how the three pillars integrate coherently.
- **Sections 6-7**: Discuss related work and future directions.
- **Section 8**: Concludes with the implications of verification-first architecture for distributed systems.

---

2. Pillar I: Bounded Verification

Bounded Verification is the foundation of this entire architecture. It defines how participants can independently validate state transitions within explicit, declared resource constraints.

This section introduces the system model (dual-ledger structure and commitment chains) and then layers formal guarantees: resource bounds, trust assumptions, verification predicates, refusal semantics, and adaptive retention.

2.1 System Model

The system comprises two complementary ledger types, each with distinct roles:

- **Account-chains (A)**: Per-account append-only logs of local state transitions. Each account maintains its own ledger, allowing parallel execution without global coordination.
- **Momentum chain (M)**: A global sequential ledger that orders commitments–cryptographic digests of account-chain states–within bounded time windows. It provides temporal anchoring and cross-account visibility without requiring full replay.

This dual-ledger separation enables verifiers to follow only the accounts they care about while maintaining cryptographic linkage to the global order.

2.1.1 Account-Chain Structure

Definition 2.1 (Account-Chain Block):
An account-chain block B_A is a tuple:

$$B_A = (h_{\text{prev}}, TX, \pi, \text{metadata})$$

where: * $h_{\text{prev}} = H(\text{previous block})$ links to the prior block, * TX is an ordered list of transactions modifying the account's state, * π is a cryptographic proof that TX represents a valid state transition, and * metadata includes auxiliary data (timestamps, signatures, app-specific fields).

Definition 2.2 (Account-Chain):
An account-chain A is a sequence of blocks:

$$A = \langle B_0, B_1, \ldots, B_k \rangle$$

where: * B_0 is the genesis block ($h_{\text{prev}} = 0$), and * for all $i > 0$: $B_i.h_{\text{prev}} = H(B_{i-1})$.

Each block must include a valid state-transition proof. Account-chains grow asynchronously–some accounts may produce thousands of blocks daily, others remain dormant for weeks.

2.1.2 Momentum Chain Structure

Definition 2.3 (Momentum Block):
A Momentum block M_i is a tuple:

$$M_i = (h_{\text{prev}}, r_C, t, \text{metadata})$$

where: * $h_{\text{prev}} = H(M_{i-1})$ links to the previous Momentum block, * r_C is the commitment root (Merkle root over the set $C = \{c_1, c_2, \ldots, c_m\}$ of account-chain commitments), * t is the consensus timestamp, and * metadata includes consensus-specific data (signatures, nonce, etc.).

Definition 2.4 (Account-Chain Commitment):
Each commitment $c \in C$ is a tuple:

$$c = (\text{addr}, h_{\text{snapshot}}, \text{height})$$

where: * addr = account address, * h_{snapshot} = hash of account state at that height, * height = account-chain height when committed.

Definition 2.4.1 (Commitment Membership Proof):
A commitment membership proof w for commitment $c \in C$ is a Merkle branch of size $O(\log m)$ hashes that proves c was included in computing r_C, where $m = |C|$ is the number of commitments in that Momentum block.

Definition 2.4.2 (Commitment Root Determinism):
The commitment root r_C is computed deterministically: commitments in C are canonically ordered (e.g., lexicographically by address, then by height) and domain-separated before Merkle root computation. This ensures all honest nodes compute identical r_C from the same commitment set.

Definition 2.5 (Momentum Chain):

$$M = \langle M_0, M_1, \ldots, M_N \rangle$$

where M_0 is the genesis Momentum block, each M_i links cryptographically to M_{i-1}, and N is the current chain height.

2.1.3 Operational Semantics

1. **Execution Phase**: Accounts process transactions independently, producing account-chain blocks containing validity proofs.
2. **Commitment Phase**: Periodically (e.g., every 10 seconds), a new Momentum block is published containing a commitment root r_C over all active accounts' latest states.
3. **Verification Phase**:
   - A verifier locates the Momentum block M_i corresponding to time T.
   - It validates the chain from M_0 (or its last known checkpoint) to M_i.
   - It requests the commitment c for account A and its membership proof w.
   - It verifies w proves $c \in r_C$.
   - It downloads the necessary account-chain blocks for A and checks them against the commitment.

Critically, verifiers need only store Momentum headers–$O(N)$ headers for N Momentum blocks (equivalently $O(N \cdot \sigma_H)$ bytes where σ_H is the Momentum-header size (in bytes))–and validate selected account-chains on demand.

2.2 Commitment Chain Properties

The Momentum chain satisfies standard blockchain properties under explicit bounds.

Property 2.1 (Commitment Finality):
Once a commitment c appears in M_i and gains k confirmations, reversal is bounded by two distinct guarantees:

- **Hash-chain tamper evidence**: Collision resistance of H ensures that any modification to a Momentum block M_j changes its hash, breaking the link $H(M_j) = h_{\text{prev}}(M_{j+1})$. An adversary can compute new hashes for an alternative history, but cannot make that alternative history match the already-committed hashes

without finding a collision/second-preimage, which is computationally infeasible under standard assumptions.

- **Consensus finality (canonization)**: The consensus mechanism bounds acceptance of alternative histories deeper than k. The probability of a chain reorganization beyond depth k is $\Pr[\text{reorg} \geq k] \leq f_{\text{consensus}}(k)$, where $f_{\text{consensus}}$ depends on the consensus model (e.g., exponentially decreasing in k for PoW under honest-majority assumptions).

Hash chaining makes tampering detectable; consensus determines which detected history becomes canonical.

Property 2.2 (Temporal Ordering):
If $c_1 \in M_i$ and $c_2 \in M_j$ with $i < j$, then c_1 precedes c_2 globally.

Property 2.3 (Bounded Storage):
Verifiers storing only Momentum headers require $O(N)$ storage in headers, where N is the number of Momentum blocks. Checkpointing or sampling techniques can reduce storage requirements, though these trade complete cryptographic verification for additional trust assumptions or probabilistic security guarantees.

These properties enable verifiers to validate commitment chains instead of replaying execution.

2.3 Trust Model and Security Assumptions

Before defining verification predicates, we clarify what the system assumes.

2.3.1 Cryptographic Assumptions

Assumption 2.1 (Collision Resistance):
The hash function H (e.g., SHA-256) is collision-resistant: finding distinct x, x' such that $H(x) = H(x')$ is computationally infeasible.

Assumption 2.2 (Proof System Soundness):
Proof systems (e.g., signatures, zk-SNARKs) are sound: producing a valid proof for a false statement is computationally infeasible.

2.3.2 Network Model

Assumption 2.3 (Eventual Connectivity):
Verifiers can intermittently synchronize with the Momentum chain.

Assumption 2.4 (No Network-Level Censorship of Momentum Blocks):
At least one honest source can provide Momentum headers.

Non-Assumption: Account-chain blocks or proofs may be unavailable. If so, the verifier returns `REFUSED_DATA_UNAVAILABLE` instead of trusting.

2.3.3 Consensus and Liveness

Assumption 2.5 (Momentum Consensus):
The consensus mechanism guarantees: * **Safety**: Honest participants agree on the chain with high probability. * **Liveness**: New Momentum blocks appear within a bounded interval Δ (an upper bound on the Momentum block interval).

The architecture is consensus-agnostic (PoW, PoS, BFT, etc.). If uncertainty exists due to forks, verifiers follow their preferred chain or refuse queries.

2.3.4 Adversarial Model

Adversary capabilities: * Can withhold data (account-chain blocks or proofs) * Can attempt invalid block creation (fails verification) * Can partition network or launch DoS attacks

Adversary limitations: * Cannot break collision resistance or forge proofs * Cannot rewrite Momentum history beyond k confirmations (subject to consensus finality assumptions) * Cannot force verifiers to accept unverified claims

An adversary may make verification impossible–but not false.

2.3.5 Explicit Trust Boundaries

Eliminated trust categories: * RPC providers: Verifiers independently validate all claims. * Sequencers and oracles: Proofs replace trust. * Execution environments: Only cryptographic attestations matter.

Minimal trust remains in: * Consensus liveness and header availability.

Explicit non-guarantees: * Proof distribution censorship and liveness under partition.

All boundaries are declared so participants make informed trust decisions.

2.4 Formal Impossibility: Unbounded Composition vs Bounded Verification

Theorem 2.1 (Composition-Verification Impossibility):
For a verifier V with resource bound R, no protocol can simultaneously: 1. Validate arbitrary compositional depth of dependent claims, 2. Guarantee correctness under adversarial data unavailability, and 3. Stay within R.

Proof Sketch. A chain of dependent claims $C_1 \rightarrow C_2 \rightarrow \ldots \rightarrow C_k$ requires recursive verification. For adversarially chosen k, verifier V must either store all proofs (violating storage bounds), re-verify recursively (violating computation bounds), or trust external sources (violating security). Therefore, at least one of the three requirements must be violated. ■

Corollary 2.1:
Bounded verification systems must either (1) refuse queries beyond declared scope or (2) accept trusted attestations. This architecture chooses (1): honest refusal.

Operational implication: A browser verifier may validate only the last two weeks of data. Older queries return `REFUSED_OUT_OF_SCOPE` instead of silently trusting an RPC.

2.5 Resource Bounds

Bounded verification explicitly quantifies the three fundamental limits under which a verifier operates:

Definition 2.6 (Resource Bound Tuple):
Each verifier V declares a resource bound $R_V = (S_V, B_V, C_V)$ where: * S_V = maximum persistent storage (in bytes) available for headers, proofs, and state fragments; * B_V = maximum network bandwidth (in bytes) available per synchronization window; * C_V = maximum local computation budget (in operations or time) per verification session.

A verifier is **correctly bounded** if it never exceeds R_V.

Bounded verification means no assumption of global completeness–verifiers validate as much as their bounds permit and refuse anything beyond.

2.5.1 Bounded Execution

Definition 2.7 (Bounded Execution):
An execution trace E is verifiable under R_V if and only if there exists a proof π_E such that:

$$\text{Verify}(\pi_E) = \text{TRUE} \quad \text{and} \quad \text{Cost}_{\text{verify}}(\pi_E) \leq C_V$$

and $|\pi_E| \leq S_V$ and $\text{BytesFetched}(E) \leq B_V$.

In other words, execution is architecturally restricted to remain within declared verification budgets.

2.5.2 Bounded State

State is bounded through adaptive pruning and checkpointing.

Definition 2.8 (State Retention Function):
Let $\rho(t)$ denote the retention policy governing stored data age. For each verifier V, the expected storage footprint satisfies:

$$\mathbb{E}[\text{StoredBytes}_V] = \int_0^\infty g_V(t) \cdot \rho(t)\, dt \leq S_V$$

where $g_V(t)$ is the storage density function (bytes per unit time) for data of age t, and $\rho(t) \in [0, 1]$ is the retention probability at age t.

This formulation ensures finite storage under probabilistic or deterministic pruning policies while maintaining verifiable history within the retention window.

2.6 Verification Predicates

Every verification operation reduces to a logical predicate over cryptographic data. A predicate evaluates to true, false, or refused depending on available information.

Definition 2.9 (Verification Predicate):

$$P(x, D, R_V) \to \{\text{TRUE}, \text{FALSE}, \text{REFUSED}\}$$

where: * x = claim being verified (e.g., “Account A sent 5 ZNN to B”); * D = set of data available to the verifier; * R_V = verifier’s declared resource bounds.

2.6.1 Evaluation Semantics

If D contains all required proof objects and their verification fits within R_V, then: * return **TRUE** if the proofs validate the claim, * return **FALSE** if the provided proofs cryptographically contradict the claim (e.g., signature/proof verification fails, Merkle root mismatch, invalid header linkage).

If required data is unavailable, out of scope, or verification would exceed R_V, return **REFUSED**.

Property 2.4 (Total Safety Under Refusal):
A refusal can never produce a false positive. Formally, if $P(x, D, R_V) = \text{REFUSED}$, then x is neither accepted nor rejected.

This is the **refusal-as-correctness** principle.

2.6.2 Predicate Composition

Let $P_1, P_2, \ldots, P_k$ be independent predicates. Composition is defined as:

$$P_{\text{all}}(x) = P_1(x) \wedge P_2(x) \wedge \ldots \wedge P_k(x)$$

Rule 2.1 (Refusal Propagation):
If any $P_i(x) = \text{REFUSED}$, then $P_{\text{all}}(x) = \text{REFUSED}$.

This ensures verifiers never infer truth from partial data.

2.7 Refusal Semantics

Refusal is an explicit and deterministic outcome, not an error. It preserves soundness by limiting verification to provable claims.

Definition 2.10 (Refusal Code Set):

$$\mathscr{R} = \{\text{REFUSED\_OUT\_OF\_SCOPE}, \text{REFUSED\_DATA\_UNAVAILABLE}, \text{REFUSED\_COST\_EXCEEDED}$$

Each refusal code maps to a distinct failure mode: * **OUT_OF_SCOPE**: The requested claim extends beyond verifier's declared history window. * **DATA_UNAVAILABLE**: Proofs missing from the network. * **COST_EXCEEDED**: Computation or bandwidth would exceed R_V.

Refusals propagate upward through compositional predicates.

Property 2.5 (Refusal Closure):
If any subpredicate returns **REFUSED**, the entire predicate returns **REFUSED**.

2.7.1 Operational Behavior

When a verifier refuses: * It emits a machine-readable refusal code and associated reason. * It records a refusal witness containing: the last verified header hash, the identifier of the missing or unverifiable object, and the bound-exceeded code. * User interfaces display “verification refused” rather than “verification failed.”

This communicates bounded correctness to users and external protocols.

2.7.2 Refusal Safety

Theorem 2.2 (Refusal Safety):
Assuming collision resistance and proof soundness, no adversary can cause a verifier to accept a false claim without its explicit consent.

Proof Sketch. All valid claims must be supported by cryptographic proofs. If proofs are unavailable, the verifier returns **REFUSED**. If forged proofs are presented, they fail verification (by proof soundness). Therefore, the only way a verifier accepts a claim is if it possesses a valid proof that passes verification. ■

This means verifiers may fail to answer–but never lie.

2.8 Adaptive Retention

Over time, state and proof data can grow without bound. Adaptive retention provides a mathematically bounded strategy for data aging.

Definition 2.11 (Adaptive Retention Policy):
Each verifier defines a retention policy $\rho(t) \in [0, 1]$ specifying the probability (or deterministic choice) of retaining data of age t. Given a storage density function $g_V(t)$ (bytes per unit time), the policy must satisfy:

$$\mathbb{E}[\text{StoredBytes}_V] = \int_0^\infty g_V(t) \cdot \rho(t)\, dt \leq S_V$$

with $\rho(0) = 1$ (all fresh data retained) and $\rho(t)$ non-increasing.

This guarantees finite storage while retaining recent data at full fidelity.

Example:

$$\rho(t) = e^{-\lambda t}$$

produces exponential decay of retention probability, keeping recent activity fully verifiable while bounding total storage.

2.8.1 Bounded Reconstruction

Property 2.6 (Bounded Reconstruction):
Any verifier can reconstruct a provable view of the system up to time $t - \Delta$, where Δ is its retention window.

Offline clients can resynchronize by re-downloading Momentum headers and requesting missing proofs within Δ.

2.9 Worked Example: Verifying a Transaction

Let's illustrate bounded verification through a single transaction example.

Scenario: A lightweight browser client wants to verify that "Account A sent 5 ZNN to B at height h."

Step 1: Obtain Momentum Headers
The client downloads the sequence of Momentum headers from M_0 to M_i. This cost is $O(N)$ headers and fits within S_V.

Step 2: Locate Commitment
From M_i, it requests the commitment $c = (\text{addr} = A, h_{\text{snapshot}}, h)$ and its membership proof w.

Step 3: Verify Membership
It verifies that w proves c is included in r_C (the commitment root in M_i).

Step 4: Retrieve Account Proofs
It requests from the network the account-chain segment $B_{h-1} \rightarrow B_h$ and the associated proof π.

Step 5: Evaluate Predicate

$$P(x, D, R_V) = \begin{cases} \text{TRUE} & \text{if Verify}(\pi, h_{\text{snapshot}}) = \text{TRUE and Cost}_{\text{verify}}(\pi) \leq C_V \\ \text{FALSE} & \text{if proofs cryptographically contradict the claim} \\ \text{REFUSED} & \text{if } \pi \text{ missing or cost exceeds } R_V \end{cases}$$

Step 6: User Feedback
The client displays: * “Transaction verifiable under current resource limits.”
or * “Verification refused – proof unavailable.”

This example demonstrates bounded correctness: verification is absolute within bounds, and gracefully refused outside them.

2.10 Operational Consequences

1. **Composable Proof Availability**: Clients can cache and share proof segments without needing trust.
2. **Proof Marketplaces**: Because proofs are verifiable objects, third parties can offer them for a fee without custody risk.
3. **Lightweight Client Inclusivity**: Devices with hundreds of megabytes of storage can still independently verify activity.
4. **Data Sovereignty**: Verifiers decide their own trust and retention policies.

2.11 Summary of Bounded Verification Properties

Property	Meaning	Guarantee
Refusal Safety	Refusals cannot produce false positives	Soundness
Bounded Storage	Verification state $\leq S_V$	Scalability
Bounded Computation	Each proof verifies in $\leq C_V$ steps	Device compatibility
Bounded Bandwidth	Proof fetch $\leq B_V$ per sync	Intermittent operation
Genesis Anchoring	Trust root exists forever	Offline recovery
Adaptive Retention	Finite storage with progressive decay	Longevity

2.12 Conclusion of Pillar I

Bounded verification redefines validation as a finite, resource-constrained, cryptographically sound process. It replaces the illusion of universal completeness with

explicitly bounded correctness. Every verifier knows precisely what it can prove, what it cannot, and when to refuse.

This foundation supports the next two pillars: * **Proof-Native Applications (zApps)** – applications whose correctness is proven cryptographically, not replayed; and * **Composable External Verification (CEV)** – trustless validation of external facts.

---

3. Pillar II: Proof-Native Applications (zApps)

This section describes a proposed extension to the base architecture: proof-native applications where correctness is established via cryptographic proofs rather than execution replay.

3.1 Motivation

Traditional smart-contract systems equate verification with re-execution: to confirm a transaction's correctness, verifiers must replay every instruction inside a virtual machine. This ties verification cost to execution complexity–an inherent scalability bottleneck.

Proof-native applications (zApps) invert that relationship. Rather than replaying computation, verifiers check succinct proofs that attest a computation's correctness. Execution can therefore occur anywhere–off-chain, asynchronously, or on heterogeneous hardware–without compromising verifiability.

3.2 Definition of a zApp

Definition 3.1 (zApp):
A zApp is an application that emits, for every state transition, a validity proof

$$\pi : \text{Compute}(\text{input}, \text{state}) \rightarrow (\text{output}, \text{state}')$$

such that

$$\text{Verify}(\pi, \text{input}, \text{state}, \text{output}, \text{state}') = \text{TRUE}$$

under the soundness assumption of its proof system.

A zApp is therefore a proof-emitting function with deterministic semantics, not an interpreted program requiring replay.

3.2.1 Execution vs Verification Separation

Layer	Role	Cost Domain
Executor	Runs arbitrary computation off-chain	Unbounded
Verifier	Checks succinct proof on-chain or locally	Bounded ($\leq C_V$)

This separation decouples expressiveness from verification overhead.

3.3 Proof Systems

zApps may use any sound proof system that satisfies verification constraints. Typical examples include:

- **zk-SNARKs** (e.g., Groth16, PLONK): constant-size proofs, fast verification.
- **STARKs**: transparent, post-quantum-secure proofs; larger but scalable.
- **Bulletproofs**: logarithmic proof size, no trusted setup.

Property 3.1 (Proof Verifiability):
Each proof system provides a deterministic verification function

$$\text{Verify}(\pi, x) \rightarrow \{0, 1\}$$

whose cost is upper-bounded by a polynomial in $\log(|x|)$.

3.3.1 Proof Object Format

Every proof object includes:

$$\pi = (\text{proof\_bytes}, \text{schema\_hash}, \text{public\_inputs})$$

The `schema_hash` ensures the verifier interprets proofs according to the correct circuit or constraint system.

3.4 zApp Lifecycle

1. **Circuit Design**: Developer defines the computation as an arithmetic circuit or constraint system.
2. **Setup**: Generates verification key vk and (if required) proving key pk.
3. **Deployment**: Publishes vk in the account-chain metadata.
4. **Execution**: Users or executors generate proofs π for state transitions.
5. **Verification**: Verifiers evaluate:

$$\text{Verify}(vk, \pi, \text{public\_inputs}) = \text{TRUE} \quad \text{and} \quad \text{Cost}_{\text{verify}}(\pi) \leq C_V$$

returning **TRUE**, **FALSE**, or **REFUSED** based on proof validity and resource constraints.

This lifecycle ensures every observable state change is accompanied by a verifiable proof.

3.5 Proof-Native State Updates

Definition 3.2 (Proof-Native Block):
A block B_z in a zApp account-chain consists of

$$B_z = (h_{\text{prev}}, \text{inputs}, \text{outputs}, \pi, \text{metadata})$$

and is valid if and only if

$$\text{Verify}(vk, \pi, \text{inputs}, \text{outputs}) = \text{TRUE}$$

Hence, state transitions are self-verifying; replay is unnecessary.

3.6 Composability

Multiple zApps can interoperate by composing their proofs.

Definition 3.3 (Composable Proof):
Given proofs $\pi_1, \pi_2, \ldots, \pi_k$ over circuits $C_1, \ldots, C_k$, a composed proof Π verifies if

$$\forall i, \text{Verify}(vk_i, \pi_i) = \text{TRUE}$$

and an aggregation circuit verifies their joint correctness.

Aggregated proofs enable efficient multi-application verification while respecting resource bounds.

3.6.1 Bounded Composability

Theorem 3.1 (Bounded Proof Composition):
For a verifier with bound C_V, there exists a maximum compositional depth $d_{\max}$ such that verification of $d > d_{\max}$ proofs exceeds C_V. Verifiers must therefore refuse deeper compositions.

Operational implication: Light clients might verify up to 5 composed proofs and refuse further nesting.

3.7 Proof-Native Application Interfaces

zApps expose deterministic, verifiable APIs:

Function	Description	Verifiability
`verifyProof(π)`	Checks proof correctness	Deterministic
`getCommitment()`	Returns Merkle root of current state	Deterministic
`requestState(height)`	Retrieves proof-backed snapshot	Bounded
`composeProofs([π₁…πₙ])`	Returns aggregated proof	Bounded by C_V

These interfaces are designed for browser-native operation and machine verification.

3.8 zApp Example: Token Ledger

A simple proof-native token ledger illustrates these ideas.

Circuit Definition

Constraints:

$$\begin{cases} \text{balance}[A] \geq \text{amount} \\ \text{balance}'[A] = \text{balance}[A] - \text{amount} \\ \text{balance}'[B] = \text{balance}[B] + \text{amount} \end{cases}$$

The proof attests that balances are updated correctly without exposing private data.

Verification

Verifiers check π against the published verification key vk_{token}. If valid, they update the account-chain state root; otherwise, they reject or refuse.

3.9 Proof-Carried Execution

In this architecture, execution artifacts themselves are proofs.

Definition 3.4 (Proof-Carried Computation):
A computation $f(x)$ is proof-carried if it emits (y, π) such that

$$\text{Verify}(vk, \pi, x, y) = \text{TRUE}$$

The verifier never re-executes f; it only validates π.

3.10 Security Model

Soundness of zApps inherits from the underlying proof system.

Theorem 3.2 (Soundness of zApps):
If the proof system is sound and the verification key is authentic, no adversary can produce π that passes verification for an incorrect computation.

Proof Sketch. Assume an adversary generates a false π. By proof-system soundness, the probability of this is negligible in λ, the security parameter. ■

Hence, correctness reduces to verifying vk authenticity.

3.11 Deployment Implications

1. **Off-Chain Scalability**: Heavy computation occurs off-chain; on-chain verification remains constant.
2. **Privacy**: zk-proofs can hide inputs while proving validity.

3. **Cross-Environment Portability**: Proofs remain valid across networks if the verification key is portable.
4. **User-Side Verification**: Browsers can independently verify proofs without RPC trust.
5. **Composable Services**: Applications can reuse proofs from others without re-execution.

3.12 Summary of Pillar II

Concept	Definition	Benefit
Proof-Native Execution	Computations emit proofs	Removes replay
Sound Verification	Constant-time validation	Bounded cost
Proof Composition	Aggregation of independent proofs	Inter-zApp composability
Resource-Aware Verification	Refusals beyond C_V	Predictable performance
Off-Chain Scalability	Heavy work moved off-chain	High throughput

zApps demonstrate how verification-first design enables expressive applications under explicit resource bounds. They form the operational layer on top of the bounded verification foundation.

---

4. Pillar III: Composable External Verification (CEV)

This section describes a proposed mechanism for independently confirming external chain events (like Bitcoin transactions) without trusting bridges, custodians, or intermediaries, using the same verification-first principles and bounded resource constraints.

4.1 Motivation

Cross-chain interoperability traditionally relies on trusted intermediaries: bridges, relayers, or custodial contracts. These entities attest that an event occurred on another chain–but such attestations require trust, not verification.

Composable External Verification (CEV) replaces trust with cryptographic evidence. A verifier can independently confirm a claim about an external system–such as "this Bitcoin transaction was confirmed"–using only public data and succinct proofs, without executing or trusting that external system.

4.2 Definition

Definition 4.1 (External Verification Claim):
A claim X about an external system S_{ext} is a tuple:

$$X = (S_{\text{ext}}, \text{statement}, \text{proof}, \text{reference})$$

where: * S_{ext} identifies the external system (e.g., Bitcoin mainnet), * statement is the factual claim (e.g., "txid T confirmed in block b"), * proof is the cryptographic proof attesting to the claim, and * reference includes consensus parameters such as block header hash.

Definition 4.2 (Composable External Verification):
A CEV mechanism is a protocol that allows internal verifiers to check external claims within bounded resources. For claim X, the verification predicate evaluates to:

$$\text{Verify}_{\text{CEV}}(X, D, R_V) = \begin{cases} \text{TRUE} & \text{if all proofs validate and costs} \leq R_V \\ \text{FALSE} & \text{if proofs cryptographically contradict the claim} \\ \text{REFUSED} & \text{if data unavailable or costs exceed } R_V \end{cases}$$

4.3 Core Idea: External Proof Composition

Instead of relaying full foreign state, the external system's consensus digest (e.g., Bitcoin block header) is imported periodically into the Momentum chain as a commitment root.

Verifiers then check inclusion proofs against that root.

Definition 4.3 (External Commitment):
An external commitment is a tuple:

$$c_{\text{ext}} = (S_{\text{ext}}, h_{\text{header}}, \text{height}, \text{metadata})$$

anchored within a Momentum block:

$$M_i = (\ldots, C \cup \{c_{\text{ext}}\}, \ldots)$$

where h_{header} is the hash of the external chain's block header at the specified height.

This anchoring ties an external chain's verified state into the internal commitment chain without requiring continuous synchronization. The anchored commitment provides header hash authenticity, while consensus confidence (e.g., that the header represents the canonical chain with sufficient finality) requires additional header-chain evidence such as PoW linkage or confirmation depth within bounded resources. Verifiers seeking stronger guarantees may validate this additional evidence; otherwise they may refuse queries if the available evidence is insufficient for their security requirements, consistent with the refusal semantics of §2.7 and the consensus-agnostic design principle.

4.4 Example: Verifying a Bitcoin Transaction

A verifier wishes to confirm that a Bitcoin transaction with hash `txid` was included in a sufficiently confirmed block at height h.

Step 1: Obtain External Commitment
A recent Momentum block M_i contains:

$$c_{\text{ext}} = (S_{\text{ext}} = \text{Bitcoin}, h_{\text{header}}, \text{height} = h)$$

where $h_{\text{header}} = H(\text{hdr}_h)$ is the hash of the Bitcoin block header at height h.

Step 2: Retrieve Proof and Headers
The verifier downloads: * Bitcoin block header hdr_h * Bitcoin SPV proof π_{SPV} consisting of: * Transaction hash `txid` * Merkle branch P (path from `txid` to Merkle root) * (Optional, for confirmation depth) subsequent headers $\text{hdr}_{h+1}, \ldots, \text{hdr}_{h+k}$

Step 3: Verify Header Anchoring
Verify that $H(\text{hdr}_h) = h_{\text{header}}$ (matching the anchored commitment in M_i).

Step 4: Evaluate Merkle Inclusion
Compute the Merkle root from the transaction:

$$\text{root}_{\text{computed}} = \text{MerkleRoot}(\text{txid}, P)$$

where MerkleRoot applies the Merkle branch P to `txid`.

Check if $\text{root}_{\text{computed}} = \text{hdr}_h$.merkle_root (the Merkle root field in the Bitcoin header).

Step 5: Verify Confirmation Depth (Consensus Evidence)
To establish that the transaction is sufficiently confirmed: * Verify the header chain $\text{hdr}_h \rightarrow \text{hdr}_{h+1} \rightarrow \ldots \rightarrow \text{hdr}_{h+k}$ by checking: * Each header's `prev_block_hash` matches H(previous header) * Each header satisfies PoW validity by checking that $H(\text{hdr}_j) <$ target(hdr_j), where target(hdr_j) is derived from the difficulty field (e.g., `nBits`) in hdr_j * If the verifier's policy requires k confirmations and the chain has valid PoW linkage for k blocks, the transaction is considered confirmed.

Step 6: Evaluate Predicate

$$P_{\text{BTC}}(\text{txid}, h, k) = \begin{cases} \text{TRUE} & \text{if } \text{root}_{\text{computed}} = \text{hdr}_h\text{.merkle\_root} \\ & \quad \text{and header chain valid for } k \text{ blocks} \\ & \quad \text{and } \text{Cost}_{\text{verify}} \leq C_V \\ \text{FALSE} & \text{if Merkle proof or header chain cryptographically invalid} \\ \text{REFUSED} & \text{if data unavailable or cost exceeds } R_V \end{cases}$$

Step 7: Output
If valid, the verifier declares the transaction confirmed under the external proof commitment with k confirmations. If the Merkle proof or header chain is cryptographically invalid, return **FALSE**. Otherwise, refuse due to unavailable or unverifiable data.

4.5 Formal Structure

Definition 4.4 (External Verification Predicate):

$$P_{\text{ext}}(X, D, R_V) = \begin{cases} \text{TRUE}, & \text{if all cryptographic checks pass and } \text{Cost}_{\text{verify}}(D) \leq C_V, \\ & \quad \text{BytesFetched}(D) \leq B_V, \text{BytesStored}(D) \leq S_V \\ \text{FALSE}, & \text{if provided cryptographic objects contradict the claim} \\ \text{REFUSED}, & \text{if data is unavailable, out of scope, or resource bounds would be exc} \end{cases}$$

This mirrors the bounded verification semantics of internal predicates (§2.6). External proofs must therefore be succinct and efficiently checkable.

4.6 Properties

1. **Trust-Minimized**: Verifiers depend only on cryptographic commitments anchored in the Momentum chain.
2. **Composable**: External verification results can themselves be inputs to other predicates (e.g., DeFi apps using BTC proofs).
3. **Bounded**: Verification cost remains within R_V.
4. **Refusal Safety**: If external data is unavailable, the verifier refuses deterministically.

4.7 External Proof Structures

External proofs depend on the external system's commitment model:

External System	Proof Type	Root Type	Verification Cost
Bitcoin	SPV (Merkle inclusion + PoW chain)	Block header hash + merkle_root field	$O(\log n_{\text{tx}})$ hashes + $O(k)$ header checks
Ethereum	Merkle Patricia proof	State root	$O(\log n_{\text{state}})$ hashes
Mina	Recursive SNARK	State commitment	$O(1)$
Other PoS/BFT Chains	Header signatures + inclusion proof	Consensus root	$O(\log n)$ hashes

4.8 Bounded Synchronization

Verifiers do not need to track all external headers. They only require recent ones anchored via CEV commitments.

Property 4.1 (Bounded Header Set):
Let H_{ext} be the external header set anchored up to height h. Verifiers need only store the last k headers satisfying

$$|H_{\text{ext}}| \leq f(S_V, B_V)$$

This ensures synchronization remains resource-bounded.

4.9 Multi-Chain Composition

CEV allows aggregation of multiple external verifications into a single proof.

Example:
A zApp verifies that: * a Bitcoin payment occurred, and * an Ethereum contract emitted a matching event.

Both predicates can be combined:

$$P_{\text{multi}} = P_{\text{BTC}} \wedge P_{\text{ETH}}$$

By Refusal Propagation (§2.6.2), if either is **REFUSED**, the entire composite predicate is **REFUSED**.

Thus, the system composes external facts without introducing cross-chain trust.

4.10 Cross-Domain Proof Aggregation

To support scalable interoperability, the system can embed aggregated proofs inside Momentum blocks.

Definition 4.5 (Aggregated External Proof):
An aggregated proof Π_{ext} combines multiple external verifications:

$$\Pi_{\text{ext}} = \{(\pi_{\text{BTC}}, vk_{\text{BTC}}), (\pi_{\text{ETH}}, vk_{\text{ETH}}), \ldots\}$$

A meta-verifier validates all subproofs under bounded cost:

$$\forall i, \text{Verify}(vk_i, \pi_i) = \text{TRUE} \quad \text{and} \quad \sum_i \text{Cost}_{\text{verify}}(\pi_i) \leq C_V$$

This produces efficient multi-chain verification that remains within client limits.

4.11 Refusal Semantics for External Data

Refusal is especially important for cross-chain claims:

Refusal Code	Meaning
`REFUSED_HEADER_MISSING`	Anchored header not found within retention window
`REFUSED_PROOF_UNAVAILABLE`	SPV or external proof not retrievable
`REFUSED_VERIFICATION_COST`	Proof size or verification cost exceeds R_V
`REFUSED_INSUFFICIENT_CONFIRMATIONS`	Header chain evidence insufficient for required confirmation depth

Refusal explicitly signals unverifiable external data, preserving system soundness.

4.12 Genesis Anchoring for External Systems

The first external commitment is embedded at genesis as a trust root. From that point, all subsequent external commitments are chained cryptographically.

Definition 4.6 (External Trust Root):

$$T_{\text{ext}} = (S_{\text{ext}}, h^{(0)}_{\text{header}}, \text{metadata})$$

All verifiable external claims must derive from T_{ext} by recursive commitments.

This ensures long-term verifiability even after long offline periods.

4.13 Example: Trustless Cross-Chain Swap

A user executes a cross-chain atomic swap between ZNN and BTC.

Sequence:

1. User locks BTC in a Bitcoin HTLC (script-based timelocked output); generates SPV proof π_{BTC} with confirmation evidence.
2. Submits π_{BTC} to a zApp on the dual-ledger system.

3. zApp verifies π_{BTC} using the latest CEV commitment and confirmation depth check.
4. Upon success, zApp releases equivalent ZNN on-chain.

No intermediary ever holds both assets–security derives purely from verification.

4.14 Security Model

Theorem 4.1 (External Verification Soundness):
Assuming: 1. external commitment anchoring is correct, and 2. external proof systems are sound,

then no adversary can produce a false verified external claim without breaking underlying cryptography.

Proof Sketch. External proofs verify against anchored roots. To falsify a claim, the adversary must either forge a valid proof (break soundness) or rewrite anchored Momentum history (break commitment immutability under Property 2.1). Both are computationally infeasible under standard cryptographic assumptions. ■

4.15 System Implications

1. **No Trusted Bridges**: CEV replaces bridges with verifiable proofs.
2. **Verifiable Oracles**: External data (e.g., prices) can be verified cryptographically.
3. **Cross-Chain Composability**: Applications can depend on multiple external proofs.
4. **Bounded Cost**: Verification remains feasible for light clients.
5. **Self-Sovereign Verification**: Any verifier can confirm external claims independently.

4.16 Summary of Pillar III

Concept	Definition	Benefit
External Commitments	Anchored roots from external chains	Trustless cross-chain linkage
External Proofs	SPV or zk proofs from external data	No replay

Concept	Definition	Benefit
Refusal Semantics	Explicit refusals for missing data	Safety under unavailability
Composability	Cross-chain proof aggregation	Interoperability
Genesis Anchoring	Permanent external trust roots	Long-term verification

Composable External Verification extends verification-first design beyond a single ledger, enabling a web of independently verifiable systems. It transforms interoperability from a problem of trust into one of cryptographic composition.

---

5. System Integration

5.1 Unified Architecture

The three pillars–Bounded Verification, Proof-Native Applications, and Composable External Verification–compose into a single, layered system:

Layer	Component	Core Function
L_1	Momentum Chain	Global commitment ordering
L_2	Account Chains	Parallel local execution and proof emission
L_3	zApps	Proof-native state transitions
L_4	CEV Interface	Verification of external systems

The architecture forms a verification-first stack:

$$\text{Execution} \subseteq \text{Proof Emission} \subseteq \text{Bounded Verification}$$

Every observable state change–internal or external–is verifiable by a resource-bounded participant operating solely on cryptographic data.

5.2 Data Flow

1. **Local Execution**: Each account runs its zApp, producing state updates and validity proofs.
2. **Commitment Inclusion**: Proof-anchored state digests are committed into the Momentum chain.
3. **External Verification**: CEV anchors external commitments (e.g., Bitcoin headers).
4. **Verification Path**:

$$\text{Verifier} \Rightarrow \text{Momentum Header} \Rightarrow \text{Commitment} \Rightarrow \text{Proof}$$

At any point, if data or cost exceed R_V, the verifier returns a refusal code.

5.3 Formal Composition

Definition 5.1 (Verification Pipeline):

$$P_{\text{system}}(x) = P_{\text{momentum}}(x) \wedge P_{\text{account}}(x) \wedge P_{\text{proof}}(x) \wedge P_{\text{external}}(x)$$

with refusal propagation (§2.6.2):

$$\exists i, P_i(x) = \text{REFUSED} \Rightarrow P_{\text{system}}(x) = \text{REFUSED}$$

This closure ensures global soundness without global execution.

5.4 Security Model

Threat	Mitigation	Source
Forged transactions	Proof-system soundness	§3.10
Header rewriting	Momentum immutability	§2.2
Data withholding	Refusal semantics	§2.7
Cross-chain forgery	Anchored external commitments	§4.14
State overflow	Adaptive retention	§2.8

Security reduces to the cryptographic primitives; no single party can induce acceptance of an invalid claim.

5.5 Resource Scaling

Let: * N = number of Momentum blocks * m = average number of commitments per Momentum block * L_A = length of account-chain segment being verified (in blocks) * σ_B = average bytes per account-chain block * σ_π = average bytes per proof object * C_{verify} = verification cost per proof (operations or time) * $R_V = (S_V, B_V, C_V)$ = verifier bounds

Then for a verifier tracking headers and selectively verifying accounts:

Storage (Momentum headers):

$$O(N) \text{ headers}$$

Bandwidth (commitment membership proof):

$$O(\log m) \text{ hashes}$$

Computation (commitment membership verification):

$$O(\log m) \text{ hash operations}$$

Bandwidth (account segment retrieval):

$$O(L_A(\sigma_B + \sigma_\pi)) \text{ bytes}$$

Computation (account proof verification):

$$O(L_A \cdot C_{\text{verify}}) \text{ operations, where } C_{\text{verify}} \text{ is bounded by } C_V$$

The commitment membership proof scales logarithmically with commitments per block; account-chain verification cost scales linearly with segment length and depends on the specific proof system used.

---

6. Related Work

6.1 Blockchain Verification Approaches

- **Full Nodes**: Execute every transaction; verification = replay.
- **Light Clients**: Validate headers only; depend on unverified intermediaries.
- **Rollups / Validity Proofs**: Off-chain execution, on-chain verification–but typically bounded to one chain and large proofs.

The proposed architecture generalizes these patterns into a dual-ledger, verification-first model with explicit resource limits.

6.2 Cryptographic Foundations

The design builds on established primitives:

Primitive	Role in Architecture
Merkle trees	Commitment membership proofs (Momentum r_C); transaction inclusion proofs (Bitcoin SPV)
SNARKs / STARKs	zApp validity proofs
Hash chains	Momentum ordering
Signature aggregation	Consensus header validation

Unlike rollups or zero-knowledge blockchains, verification cost here is tunable via R_V, not fixed to a single circuit.

6.3 Conceptual Lineage

- **Bitcoin SPV (2009)**: Introduced header-only verification.
- **Plasma (2017)**: Proposed bounded off-chain state commitment.
- **zkRollups (2018+)**: Shifted execution to proofs.
- **Celestia (2022)**: Modularized consensus and data availability.
- **Mina Protocol (2023)**: Emphasized succinct chain proofs.

This work synthesizes their insights into a single verification-bounded framework.

7. Discussion and Implications

7.1 Refusal as a Primitive

Traditional systems treat unresponsiveness as failure. Here, refusal is correctness:

$$\text{Unverifiable} \Rightarrow \text{Refuse, not Trust}$$

This principle converts resource limitations into formally safe boundaries.

7.2 Verifiable Sovereignty

Each verifier defines its own trust radius and resource policy. The network becomes a federation of independently verifying participants, not a monolithic consensus on execution.

This supports lightweight nodes, mobile wallets, and intermittent connectivity without delegation.

7.3 Sustainability and Longevity

Adaptive retention limits data growth; verifiers can remain functional indefinitely with finite storage. Genesis anchoring guarantees recovery after long offline periods–critical for archival and planetary-scale participation.

7.4 Open Research Questions

- Efficient proof markets for decentralized proof distribution.
- Recursive composition limits under tight C_V bounds.
- Post-quantum-secure proof systems optimized for browsers.
- Incentive design for external commitment publication.

These remain active areas for community research.

8. Conclusion

Bounded verification transforms distributed systems from execution-first to verification-first architectures. By separating execution, proof generation, and verification, and by bounding each verifier's resources, the design achieves three properties simultaneously:

1. **Independent Verification**: Any participant can verify correctness without trust.
2. **Bounded Scalability**: Verification cost is explicit, predictable, and device-feasible.
3. **Composable Trust**: Internal and external proofs integrate seamlessly.

This model redefines what it means for a distributed ledger to be trustless. It replaces implicit completeness with explicit verifiability and explicit refusal. In doing so, it lays the groundwork for a sustainable, universally accessible cryptographic economy.

Appendix A: Reader's Guide

A.1 Intended Audiences

- **Protocol Designers / Cryptographers**: Formal predicates and proofs.
- **Implementers**: Algorithms, interfaces, and resource accounting.
- **Conceptual Readers**: Operational overviews and analogies.

A.2 Reading Paths

Level	Sections	Approx. Time
Conceptual Overview	§§1, 2 (skim), 3.1-3.3, 8	≈ 2 h
Technical Understanding	All prose + examples	≈ 5 h
Formal Analysis	Full definitions + proofs	8 h+

A.3 Notation

Mathematical symbols follow standard cryptographic conventions:

- $H(\cdot)$ = cryptographic hash function
- π = proof
- vk = verification key
- $R_V = (S_V, B_V, C_V)$ = verifier resource bounds
- N = number of Momentum blocks
- m = number of commitments per Momentum block
- r_C = commitment root (Merkle root over commitments)

Appendix B: Refusal Codes

Code	Meaning	Example Cause
`REFUSED_OUT_OF_SCOPE`	Query exceeds declared history window	Requesting data > Δ
`REFUSED_DATA_UNAVAILABLE`	Missing proof or account segment	Peer offline
`REFUSED_COST_EXCEEDED`	Computation > C_V	Excessive proof aggregation
`REFUSED_HEADER_MISSING`	External header unavailable	External anchoring gap
`REFUSED_PROOF_UNAVAILABLE`	SPV or external proof not retrievable	SPV data unavailable
`REFUSED_VERIFICATION_COST`	Proof size exceeds R_V	Oversized proof
`REFUSED_INSUFFICIENT_CONFIRMATIONS`	Header chain evidence insufficient for required confirmation depth	Insufficient PoW confirmations

---

Appendix C: Glossary

Term	Definition
Momentum Chain	Sequential ledger of commitments providing global order.
Account Chain	Per-account ledger enabling parallel execution.
zApp	Proof-native application producing verifiable state transitions.
CEV	Composable External Verification; mechanism for verifying external systems.
Bounded Verification	Validation limited by explicit resource constraints.
Refusal	Deterministic rejection when proof cannot be verified within bounds.

Appendix D: Notation Summary

Symbol	Meaning
$R_V = (S_V, B_V, C_V)$	Resource bounds (storage, bandwidth, computation)
$P(x, D, R_V)$	Verification predicate
$\rho(t)$	Retention function
c_{ext}	External commitment
T_{ext}	External trust root
vk, π	Verification key / proof
$H(\cdot)$	Cryptographic hash function
N	Number of Momentum blocks
m	Number of commitments per Momentum block

Symbol	Meaning
r_C	Commitment root (Merkle root over commitments)
L_A	Length of account-chain segment (in blocks)
σ_B	Average bytes per account-chain block
σ_π	Average bytes per proof object
C_{verify}	Verification cost per proof (operations or time)
$\text{Cost}_{\text{verify}}(\pi)$	Cost function returning verification cost for proof π
$\text{BytesFetched}(D)$	Total bytes downloaded for data set D
$\text{BytesStored}(D)$	Total bytes retained for data set D

Appendix E: Data Availability and Proof Distribution

This appendix makes the data-availability (DA) and proof-distribution assumptions explicit for Zenon's current architecture (Phase 0 / Alphanet) and for the verification-first model used throughout this paper. The goal is not to claim perfect availability, but to specify: (i) *who* is expected to store and serve which objects, (ii) *how* clients retrieve them, and (iii) the exact failure modes (including refusal).

E.1 Objects that must remain available

For a verifier to answer a query without refusal, the following objects must be retrievable from some peer set:

- **Momentum headers** (and the commitment root r_C for each Momentum): required to anchor account-chain segments and commitment-membership proofs.
- **Account-chain segments** for the queried address over the requested history window Δ.
- **Commitment membership proofs** (Merkle branches) for commitments under r_C.
- **zApp proof objects** π and verification keys *vk*, when a query depends on proof-native application claims.
- **External header chains** (e.g., Bitcoin headers) and any required inclusion proofs, when evaluating external predicates.

E.2 Who stores what (Phase 0 / Alphanet roles)

Zenon defines several staked roles that are explicitly designed to retain and relay state and history:

Role	Primary responsibility (DA / distribution)
Pillars	Produce Momentums and, as full nodes, retain and archive the entire ledger history.
Nodes	Full archival nodes that store and share the ledger and passively validate state.
Sentinels	Network participants that are registered on-chain and rewarded; commonly used as relays and availability peers for constrained clients.

Operational reading: **Pillars** and **Nodes** are the default archival substrate. **Sentinels** improve reachability for constrained clients but do not replace archival incentives.

E.3 Retrieval paths for bounded verifiers

A bounded verifier (browser / mobile) is expected to use a layered retrieval strategy:

- **Fast path:** fetch the latest Momentum header sequence (or a recent verified frontier) from any available peer set; verify producer/consensus validity per the current rules.
- **Query path:** request the minimum account-chain segment and the corresponding commitment-membership proof(s) needed to answer the query.

- **Proof path:** if the query depends on π, retrieve π and *vk* (or a hashed reference to *vk* already committed under r_C).
- **External path:** for CEV predicates, retrieve the relevant external header-chain suffix and required inclusion proofs.

E.4 Adversarial availability and explicit refusal

If any required object is unavailable within the verifier's bandwidth bound B_V (or outside its declared history window Δ), the correct response is explicit refusal rather than an unverifiable answer. The refusal codes in Appendix B correspond to the following DA failure classes:

Failure class	Refusal surface
Outside declared window Δ	`REFUSED_OUT_OF_SCOPE`
Missing proof / segment	`REFUSED_DATA_UNAVAILABLE`
Missing external headers	`REFUSED_HEADER_MISSING`
Missing external inclusion proof	`REFUSED_PROOF_UNAVAILABLE`
Exceeds resource bounds	`REFUSED_COST_EXCEEDED` (or `REFUSED_VERIFICATION_COST` for external proofs)

E.5 Minimum-viable DA commitments (recommended)

To reduce "availability hand-waving" while staying consistent with Zenon's Phase 0 roles, a minimum-viable DA posture is:

- **Pillar archival expectation:** Pillars are treated as archival by default; clients should assume Momentum headers and recent account segments are widely replicated.
- **Node archival expectation:** Nodes act as additional archival replicas and improve long-horizon availability.
- **Redundancy principle:** verifiers should request objects from k distinct peers ($k \geq 2$) before refusing, subject to B_V.
- **Content-addressing:** proof objects and large artifacts should be hash-addressed (or committed) so that any mirror can serve them without trust.

Appendix F: Economic Layer (Phase 0 / Alphanet)

This appendix summarizes the on-chain incentive structure that supports liveness and data retention in Phase 0. It is not a complete equilibrium analysis, but it grounds the roles referenced in Appendix E in concrete, implemented mechanisms.

F.1 Emission schedule (daily distribution)

Zenon distributes fixed daily rewards on a 24-hour cadence:

Asset	Daily distribution
ZNN	4,320 ZNN / 24h
QSR	5,000 QSR / 24h

These emissions are the primary budget used to reward consensus production, staking, and availability roles.

F.2 Pillar incentives

Pillars participate in consensus by producing and validating Momentums, and they are explicitly described as full archival nodes. Pillar rewards are split into:

- **Momentum-interval rewards:** proportional to the number of Momentums produced during the reward period.
- **Delegation rewards:** distributed proportionally to delegated stake, subject to the Pillar's configured sharing percentages.
- **Uptime coupling:** if a Pillar produces fewer than its expected Momentums, its rewards decrease proportionally.

F.3 Registration costs and "skin in the game"

Phase 0 uses explicit stake and burn mechanics to bound Sybil participation:

- **Pillar stake:** 15,000 ZNN required to register a Pillar (refundable on disassembly); QSR is burned to create the Pillar slot and is not refundable.
- **Dynamic QSR registration cost:** the chain exposes the current registration cost via the embedded Pillar contract (e.g., `embedded.pillar.getQsrRegistrationCost`).
- **Node stake:** Nodes (archival full nodes) require staked ZNN and QSR and are rewarded when eligible.
- **Sentinel stake:** Sentinels are registered on-chain and can receive rewards; they are commonly used to improve relay availability for light clients.

F.4 Who pays for proof generation?

In Phase 0, “proof generation” is not a single global market: many proofs are produced by the party that benefits from the claim (e.g., a zApp operator, bridge relayer, or client). The protocol-level economic primitive available today is: (i) rewards for maintaining consensus/availability roles (Pillars, Nodes, Sentinels) and (ii) content-addressed commitments under Momentum roots so that third parties can mirror proofs without changing trust assumptions.

Appendix G: Consensus Instantiation and Timing (Phase 0 → Phase I)

The main body of this paper treats consensus abstractly via $f_{\text{consensus}}(k)$. This appendix instantiates the abstraction with Zenon's current Phase 0 consensus, and summarizes the Phase I roadmap direction.

G.1 Phase 0 (Alphanet) consensus summary

Zenon Phase 0 uses a delegated Proof of Stake (dPoS) approach where a set of staked nodes ("Pillars") take turns producing Momentum blocks on a strict schedule.

- **Scheduled production:** time is divided into short slots ("ticks"); one Pillar is assigned per tick to produce the next Momentum.
- **Weighted selection:** higher-staked Pillars produce more frequently; selection draws from (i) the top 30 by delegated weight and (ii) additional Pillars outside the top 30 with lower frequency.
- **Archive expectation:** Pillars are treated as full nodes that retain and archive the entire ledger history.

G.2 Momentum interval parameter

The published target Momentum interval is 10 seconds (rewards are commonly presented per minute or per day). This parameter is the key bridge between abstract "step counts" and real-time latency in the verification and availability bounds.

G.3 Finality and reorg considerations

Phase 0's scheduled producer model reduces proposer conflicts, but practical finality still depends on fork-choice rules and network conditions. For external verification (e.g., Bitcoin headers), choose the confirmation depth k conservatively to tolerate transient forks in either system. The refusal codes `REFUSED_INSUFFICIENT_CONFIRMATIONS` and `REFUSED_HEADER_MISSING` are the verifier-visible surface for these finality gaps.

G.4 Phase I direction (Narwhal & Tusk)

Zenon documentation describes a roadmap transition toward a high-performance, leaderless consensus influenced by Narwhal & Tusk, separating transaction dissemination from ordering to improve throughput and resilience under asynchrony. This direction is consistent with the dual-ledger separation emphasized in the main paper.

Appendix H: Resource Budgets and Practical Defaults

This appendix translates the abstract verifier bounds $R_V = (S_V, B_V, C_V)$ into concrete, implementation-friendly budgets. The values below are **recommended starting points** for builders; they should be measured and tuned for target devices.

H.1 Suggested baseline budgets

Profile	S_V (storage)	B_V (bandwidth per query)	C_V (compute per query)
Browser-light	≤ 256 MB local (indexed headers + small cache)	≤ 2–5 MB	≤ 250 ms (single-thread)
Mobile-light	≤ 128 MB local	≤ 1–3 MB	≤ 300 ms
Desktop-light	≤ 512 MB local	≤ 5–10 MB	≤ 150 ms

H.2 How to account for C_V (verification cost)

A practical verifier should treat C_V as a "budget envelope" over a small set of primitives:

- **Hashing:** SHA-256 / Blake2 (Merkle branches, header chaining).
- **Signature verification:** producer signatures and transaction signatures (where applicable).
- **SNARK verification:** verifying π against *vk* (bounded by circuit choice).
- **Parsing / decoding:** bounded by message sizes that are already constrained by B_V.

Implementers should instrument end-to-end query verification time and expose it as `Cost_verify(π)` in logs to support empirical tuning.

H.3 Refusal-rate measurement protocol (recommended)

To prevent "theoretically correct but unusable" configurations, clients should measure refusal rates under realistic conditions:

- Fix (S_V, B_V, C_V) and Δ for a device profile.
- Replay a representative workload (wallet history, token transfers, CEV queries).
- Report the fraction of queries returning each refusal code (Appendix B) and the median verification time for successful queries.

Appendix I: Comparative Notes (Verification, Proving, DA)

This appendix provides a compact comparison along the axes most relevant to a verification-first design: what must be *verified* by clients, who bears *proving* costs, and what DA assumptions remain.

System	Client verification	Proving / execution cost	DA assumption (simplified)
Zenon (this paper)	Bounded verification; explicit refusal when bounds or data are missing.	Proof generation borne by claimants / operators; consensus roles rewarded on-chain.	Availability via rewarded roles (Pillars /Nodes/Sentinels); refusal on missing data.
zkSync / StarkNet (zk-rollups)	Verify validity proofs + data commitments.	Heavy proving by sequencers/relayers; throughput and latency depend on batching.	DA published to L1; censorship resistance depends on L1 inclusion.
Celestia-style DA layer	Verify DA sampling / headers.	Execution/proving off-chain on rollup chains; DA posts to the DA layer.	DA guaranteed by the DA layer; applications must publish data there.
Mina (succinct chain)	Verify recursive proof of chain state.	Recursive proof generation by block producers.	State availability still needed for user-level proofs; often relies on archival services.

I.1 Interpretation for Zenon builders

Zenon’s distinguishing choice is not “DA solved” but “DA made explicit”: when data or proofs are unavailable within declared bounds, the system returns refusal instead of silently degrading trust assumptions. This makes the availability layer a first-class engineering target (peer selection, replication, archival incentives) rather than an implicit externality.

I.2 Source pointers (non-exhaustive)

For implemented Phase 0 parameters and roles, see: Zenon’s public site (daily distribution), the Alphanet specification note (roles, target Momentum interval), and Zenon documentation pages (consensus and embedded contracts).

Appendix J: Contributions and Novelty Statement

This greenpaper is intentionally conservative in its primitives (SPV, membership proofs, validity proofs, and light-client patterns). Its contribution is the unified verification-first interface: explicit resource bounds, explicit refusal surfaces, and a single model that spans internal commitments and external facts.

J.1 Claimed contributions (summary)

C1. Verification-first semantics: A verifier model parameterized by explicit resource bounds R_V = (S_V, B_V, C_V) with refusal as a correctness outcome (not a failure).
C2. Dual-ledger split as a verification interface: Parallel execution is separated from sequential commitment ordering, so clients verify commitments rather than replay full execution.
C3. Composable External Verification (CEV): A uniform predicate interface for external facts (e.g., Bitcoin SPV) with explicit refusal surfaces for missing headers/proofs.
C4. Bounded composability: A formal limit on verification depth under fixed budgets, enabling engineering-time budgeting rather than implicit “infinite composability” assumptions.

Appendix J (cont.): Novelty framing

J.2 Synthesis vs. incremental contribution

To avoid over-claiming novelty, the table below separates well-known building blocks from the incremental value of this paper: formalizing them under R_V and refusal semantics so implementers can reason about safety, liveness, and usability under constrained clients.

Prior art	**Where it appears**	**This paper's incremental contribution**
SPV / header-chain verification	Bitcoin light clients, SPV literature	Formalized as a CEV predicate with explicit refusal codes and budget accounting under R_V.
zk-rollups / validity proofs	zkSync, StarkNet, etc.	Separated proving vs. verifying roles, and integrated verification into a refusal-first client model.
Light clients / checkpoints	Many chains	Unified “resume from local trust root” with adaptive retention and explicit out-of-scope refusal.
Dual-ledger or layered designs	Various L1/L2 hybrids	Treated as a deterministic verification interface: commitment ordering is primary; execution is secondary.

Appendix H (Addendum): Benchmark Harness and Target Budgets

To close the gap between formal guarantees and deployment reality, the project should publish a minimal, reproducible verifier harness. The harness is not a full node: it is a collection of deterministic test vectors and microbenchmarks that measure costs corresponding to (S_V, B_V, C_V) and refusal rates under retention policies.

H.A Harness outline (reproducible)

- **Test vectors:** fixed Momentum header sequences, commitment trees, account-chain segments, and CEV external headers/proofs.
- **Workloads:** (i) header sync/verify, (ii) membership verification, (iii) CEV verification, (iv) representative zApp proof verification.
- **Reporting:** median + p95 timings, peak memory, bytes fetched, and refusal counts by code.
- **Environments:** Chromium desktop, mid-range phone browser, and a low-power single-board node (optional).

H.B Initial target budgets (design, to be validated)

Workload	Metric	Target (design)	Measurement notes
Momentum header sync (N headers)	headers/sec; p95	Desktop: 5k-20k/sec Phone: 1k-5k/sec	Hash + signature verification only; run in JS/WebAssembly; report median and p95.
Commitment membership proof (Merkle branch)	verify time; bytes	<1 ms; O(log m) hashes	Measure on typical m; include deserialization cost.
CEV (BTC header suffix + SPV proof)	end-to-end verify	<50 ms (cached headers) <200 ms (cold)	Cache model: last K headers retained; cold path includes download within B_V.
SNARK verification (representative zApp proof)	verify time; memory	Class A: <250 ms Class B: <1 s	Measure chosen proof system; include wasm constraints; report failures as REFUSED_COST_EXCEEDED.
Refusal rate under retention policy	% queries refused	<1% (steady-state target)	Simulate partitions + retention windows; report by refusal code class.

Appendix H (Addendum): Proof budget classes

The phrase “browser-native verification” should be interpreted as: verification is feasible for proofs whose declared class fits within the verifier’s C_V budget. zApps should declare a proof class; verifiers refuse when the class exceeds their budget.

Class	Verification budget	Intended environment	Action if exceeded
A	C_V ≤ 250 ms; small memory	Browser / mobile default	REFUSED_COST_EXCEEDED
B	C_V ≤ 1 s; moderate memory	Mobile with batching; desktop browser	REFUSED_COST_EXCEEDED or batch
C	Unbounded / delegated	Native node / delegated verifier	REFUSED_OUT_OF_SCOPE unless delegation policy permits

Appendix E (Addendum): Data Availability Security Properties

Appendix E makes DA explicit via refusal. To reduce “DA hand-waving,” DA can be elevated to named security properties that are testable and implementable incrementally.

E.A DA-Detectability (mandatory)

Definition (DA-Detectability). For any query q whose verification requires an object o (proof, segment, header suffix), an honest verifier operating under (B_V, Δ) either (i) retrieves o from the serving layer within its bounds, or (ii) returns an explicit refusal code indicating which dependency is missing (e.g., REFUSED_DATA_UNAVAILABLE, REFUSED_HEADER_MISSING). No silent degradation of trust is permitted.

E.B DA-Retrievability (recommended target)

Definition (DA-Retrievability). Under a stated redundancy model (e.g., k independent peers, erasure coding with rate r), the probability of retrieving any required object within bounds exceeds 1-ε for typical workloads. This is an engineering target, validated by measurement (refusal-rate benchmarks) rather than assumed.

E.C Minimum cryptographic enforcement path

- **Content addressing:** proofs/segments are referenced by hash (or committed under r_C) so any mirror can serve them without additional trust.
- **Redundant querying:** verifiers request from k ≥ 2 distinct peers before refusing, subject to B_V.
- **Availability attestations (optional):** nodes/pillars publish signed “I have blob X” claims; misbehavior can be audited and penalized by policy.
- **Erasure-coded publication (optional):** large artifacts are encoded into chunks; sampling-based audits detect withholding with high probability.

Appendix F & 3 (Addendum): Proof Economics and Practical Bounded Composability

Bounding verification without addressing proving/serving incentives can create a two-tier system. This addendum specifies implementable payment models and connects Theorem 3.1 to a usable budgeting rule for composition depth.

F.A Who pays for proving? (three viable models)

- **User-pays:** the claimant submitting a zApp result pays for proof generation and inclusion (simple; aligns costs with demand).
- **Sponsor-pays:** the zApp operator subsidizes proving to reduce user friction; costs recovered via app-level fees.
- **Market provers:** independent provers bid to generate proofs; a fee market clears based on proof complexity and latency.

Recommended default for Phase 0: user-pays with optional sponsor subsidies. Fees should be parameterized by (i) proof byte size, (ii) declared proof class, and (iii) worst-case verifier cost envelope, so spam scales with the burden it imposes on bounded verifiers.

3.A Corollary (engineering bound for d_max)

Let each additional composition layer consume at least (ΔS, ΔB, ΔC) of storage, bandwidth, and computation for a target verifier class. Then a conservative bound is:

Corollary. $d_{max} \le \min(\text{floor}(S_V/\Delta S), \text{floor}(B_V/\Delta B), \text{floor}(C_V/\Delta C))$.

3.B Worked example (mobile verifier)

Example (illustrative, to be validated by the harness): suppose a mobile verifier declares C_V = 250 ms and B_V = 2 MB for a query. If one composition layer requires (i) one membership proof verification (~1 ms), (ii) a cached external header check (~20 ms), and (iii) one Class A proof verification (~150 ms), then $\Delta C \approx 171$ ms and d_{max} is at most 1 under this budget. Higher-depth composition would require batching, delegation, or a larger C_V class.