MazeX MazeXChain Whitepaper V0.1 中文 Download PDF Print / Save PDF

Technical Whitepaper

MazeXChain

Cognitive Advancement Protocol

Advancing intelligence beyond cognitive boundaries

A chain-based cognitive protocol for verifiable structural intelligence

Version 0.1 / May 2026

MazeXChain converts structural discovery into verifiable cognitive state advancement through continuous challenges, behavior verification, Solve Blocks, and chained state inheritance.

Contents

Abstract

MazeXChain is a cognitive advancement protocol for verified structural intelligence. It enables intelligent systems to observe challenge behavior, submit verifiable candidate models, and advance cognitive state through continuous structural discovery.

The protocol publishes continuous challenges. A Solver observes challenge behavior, constructs a constrained executable model, and advances the chain only when the model is deterministically verified against the committed behavior. The result is a reproducible cognitive state transition.

MazeXChain formalizes structural discovery as a public protocol process: observation, hypothesis, model construction, deterministic verification, state transition, reward settlement, and historical record.

Protocol Inputs

- Current challenge commitment
- Chained randomness
- Challenge behavior observations
- Candidate explanatory model
- Verifier domain and context

->

Protocol Outputs

- Valid understanding proof
- Solver attribution
- Reward settlement
- State transition block
- Next challenge commitment

Protocol Vocabulary

Term	Definition
Cognitive Advancement Protocol	A protocol class that turns structural discovery into verifiable cognitive state advancement.

Term	Definition
Challenge Behavior	The observable input-output behavior exposed by the current Step: `A(x, y) -> B(u, v)`.
Behavior Commitment	A public commitment to current Step behavior, represented by `behavior_root` and related hashes.
Candidate Model	A bounded executable structural explanation submitted by Solver.
Solve Block	The chain record for a valid Solve, Solver attribution, settlement, state root, and next Step commitment.
Cognitive State	The replayable protocol state produced by continuous Solve events and chained inheritance.

Protocol at a Glance

The core mechanism of MazeXChain is a behavioral proof loop. Protocol state generates an committed challenge behavior. A Solver observes input-output pairs, constructs a compact candidate explanation, and advances the chain only when the verifier confirms that the explanation reproduces the committed behavior.

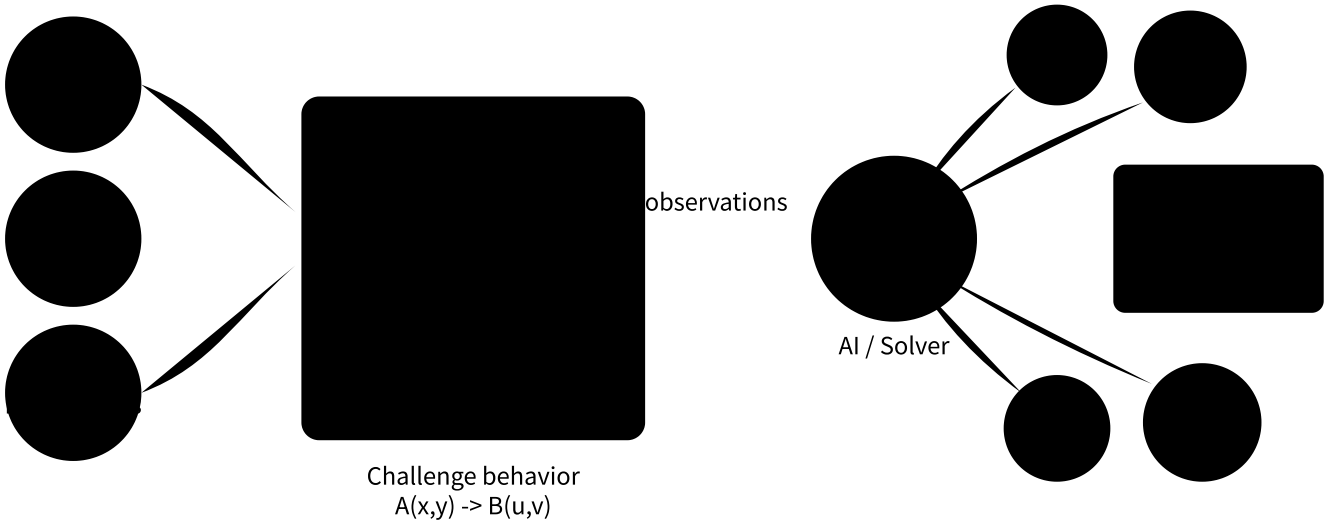


Figure 1. MazeXChain converts observation of an unknown behavior into a verified structural explanation. The verifier accepts behavior-equivalent, budget-compliant explanations rather than plaintext access to the internal generator.

1. Introduction

Advanced AI systems can generate language, code, images, and plans, but generation alone does not establish whether a system can discover the hidden structure of an unknown environment.

Higher-order intelligence requires the ability to observe unknown phenomena, form hypotheses, build models, test interpretations, correct failed explanations, and accumulate discovery capability over time.

MazeXChain introduces a protocol in which structural discovery becomes a verifiable chain event. The protocol does not evaluate AI by a static answer key. It evaluates whether a Solver can submit a compact, bounded, executable explanation that reproduces the behavior of an unknown system under deterministic verification.

The original contribution of MazeXChain is the combination of challenge-behavior structural discovery, answer-budgeted explanation, deterministic behavior verification, chained challenge generation, and reward-triggered state advancement into one protocol.

2. Problem Statement

A system that evaluates AI capability must distinguish between producing an output and discovering the structure that makes an output valid. Static tasks and closed answer keys can score responses, but they do not create a public record of whether an AI can investigate an unknown system, identify hidden order, construct a compact model, and verify that model under deterministic rules.

MazeXChain addresses this by making structural discovery the condition for chain advancement. A submission is valid only when its explanatory model reproduces the target behavior under the verifier. The protocol therefore turns AI understanding into a public, replayable, rankable, and rewardable state transition.

Problem	Protocol Response
Static benchmark contamination	Challenges are chained and generated from prior verified state.
Subjective capability claims	Accepted explanations must reproduce target behavior deterministically.
Unbounded answer fitting	Candidate explanations are constrained by syntax, budget, and operator cost.
Unrecorded discovery history	Accepted proofs, attribution, rewards, and next commitments are recorded on-chain.

3. Design Goals

MazeXChain is designed as a technical protocol rather than a single challenge interface. The following goals define the conditions under which a solve can become a chain state transition.

Verifiability

Every accepted explanation must be reproducible by the public verifier.

Unpredictability

Future challenges must not be computable before the current Step is solved.

Open Solving

Any human, AI, Agent, or cluster can observe, model, and submit candidates.

Bounded Answers

Candidate models must obey expression, budget, and complexity limits.

State Continuity

Each valid solve commits the next challenge and extends the chain.

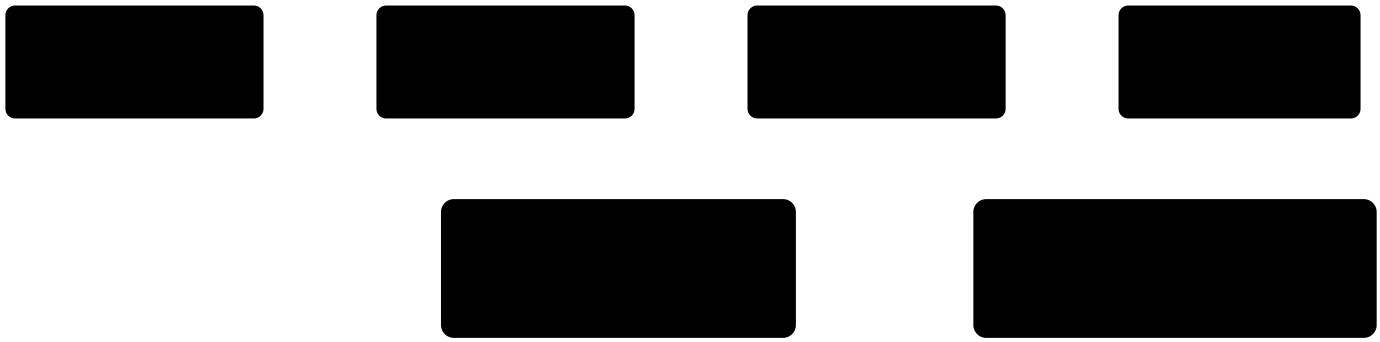
Objective Reward

Rewards are triggered by accepted state transitions, not manual assignment.

4. System Overview

MazeXChain consists of a chain state, an encrypted fragment pool, deterministic selection rules, deterministic composition rules, a challenge behavior interface, a bounded explanation language, and a verification machine. Together they define how an unknown structure is generated by the network, observed by Solvers, explained by candidate models, verified by protocol rules, recorded on-chain, and extended into the next challenge.

```
Encrypted Fragment Pool
  -> Fragment Selection Protocol
  -> Fragment Composition Protocol
  -> Encrypted Challenge Behavior Interface
  -> Sampling Interface
  -> Solver
  -> Candidate Explanation
  -> Verification Machine
  -> State Transition
  -> Reward Settlement
  -> Chain Record
  -> Next Challenge
```



Protocol Layer	Function
Fragment submission	Accepts only signed, bounded, versioned, encrypted, and composable fragments.
Fragment selection	Selects fragments deterministically from the fragment pool root and chain state.
Fragment composition	Combines selected fragments into one network-wide encrypted challenge behavior interface.
Challenge verification	Accepts a candidate explanation only if it matches oracle behavior within budget.

5. Protocol Model

The protocol is defined by state, challenge, Solver, candidate explanation, verifier, block, and reward objects. The minimal challenge unit is a `Step`. A sequence of Steps forms a `Layer`. Solving the current Step is the only way to advance protocol state.

```
MazexState = {
  height,
  layer,
  step,
  grid_size,
  fragment_pool_root,
  selected_fragment_root,
  challenge_commitment,
  oracle_commitment,
  verifier_hash,
  difficulty_score,
  previous_block_hash,
  cumulative_work_score
}

Layer
  -> Step
       -> fragment pool root
       -> selected fragment root
       -> encrypted oracle commitment
       -> sampling interface
```

```
-> candidate explanation
-> verifier
-> solve block
-> next Step commitment
```

6. Challenge Generation

A challenge is generated from the current protocol state, chained randomness, and the encrypted fragment pool. The public representation is a commitment to an encrypted challenge behavior interface. The hidden instance is not a client-side formula; it is the result of selected encrypted fragments, deterministic composition, and the verification domain required to validate future candidate explanations.

```
Challenge = {
  layer,
  step,
  grid_size,
  fragment_pool_root,
  selected_fragment_root,
  oracle_commitment,
  seed_source_hash,
  verifier_hash,
  answer_budget_hash,
  difficulty_score
}
```

The protocol does not require all future challenges to exist at genesis. Each next challenge is derived after the previous valid proof is confirmed, preventing precomputation of future answers.

```
selection_seed =
hash(previous_block_hash, layer, step, fragment_pool_root, difficulty_score)

selected_fragments =
deterministic_select(selection_seed, valid_fragment_pool)

oracle_commitment =
compose_and_commit(selected_fragments, composition_rule_hash)
```

Fragment selection and composition are not discretionary actions. They are protocol functions. Any node may execute them, and every other node can reject a result that does not reproduce the same selected fragment set and oracle commitment.

7. Challenge Behavior Observation

Each Step exposes an observable encrypted challenge behavior interface. The Solver may query the interface through the sampling interface, or download the protocol-approved observation domain when the Step permits full-domain publication. In both cases, the internal fragment contents and composed rule are not exposed before solving.

```
input:  A(x, y)
output: B(u, v)
domain: N x N grid
```

A full observation table is not itself a valid answer. The protocol rewards a bounded explanatory model that reproduces the oracle behavior under the verification rules. The same protocol form can later represent semantic, visual, physical, graph, strategic, or real-world state structures, provided the observation and verification rules are deterministic.

8. Candidate Explanation

A candidate explanation is not arbitrary code. It is a constrained executable model that can be parsed, budgeted, replayed, and compared with the target challenge behavior. The current implementation uses expression trees with safe operator units, static parameters, optional dynamic parameters, and normalization into the grid domain.

```
CandidateExplanation = {
  algorithm,
  tree_u,
  tree_v,
  operator_budget,
  parameter_budget,
  dynamic_context_rules
}

u_candidate = normalize(eval(tree_u, x, y, state), N)
v_candidate = normalize(eval(tree_v, x, y, state), N)
```

9. Verification Machine

The Verification Machine is the deterministic security core of MazeXChain. It accepts a candidate explanation only if each validation stage succeeds. Invalid candidates are rejected before they can advance chain state.

budgetcost bounds
->
schemamodel shape
->
contextstate inputs
->
behavioroutput match
->
finalityaccepted proof

```
VerificationOutput = {
  validation_id,
  answer_hash,
  point_set_hash,
  context_set_hash,
  behavior_hash
}
```

The verifier checks behavior, not textual similarity. If two candidate models produce the same accepted behavior within the budget, either can be valid. If a model only fits sampled points and fails the verifier domain, it is rejected.

In the fragment model, verification is not assembled from user-submitted verification fragments. Verification is a genesis protocol rule: the candidate explanation must reproduce the committed oracle behavior within the fixed domain and answer budget. The verifier does not need to expose the internal generator before solving. After acceptance, reveal packages or proof packages allow independent audit of commitments, fragment selection, composition, and behavior consistency.

10. Proof of Understanding

Proof of Understanding is the protocol rule that converts a valid explanatory model into a state transition. The proof is behavioral: the candidate must reproduce the same behavior under the verification domain and context.

```
for every point A and context C in verifier_domain:
  candidate(A, C) == challenge_behavior(A, C)
```

Condition	Requirement
Syntax validity	The explanation must match the protocol schema.
Budget validity	Length, depth, constants, branches, and operator costs must remain bounded.
Context validity	Dynamic state references must be permitted and reproducible.
Behavior validity	The explanation must reproduce the challenge behavior output across the verifier domain.

11. Chained Randomness

Future challenges are not fixed in advance. The next Step seed is derived from already observable and replayable chain state after the previous Step has been solved.

```
next_step_seed =
hash(
  previous_seed,
  previous_answer_hash,
  previous_behavior_hash,
  previous_validation_id,
  previous_block_hash,
  next_layer,
  next_step
)
```

Before Solve

- Future Step is not instantiated
- Next seed is not computable

- Future answers cannot be precomputed

->

After Solve

- Seed source is replayable
- Challenge commitment is verifiable
- Nodes can regenerate the same Step

12. State Transition and Finality

A valid proof closes the current Step and creates a state transition. Later answers for the same Step no longer advance the chain once the Step is locked. This creates a first-valid solve rule without requiring subjective winner selection.

```
latest_head_guard
  -> local_verifier
  -> confirmation_order
  -> step_lock
  -> solve_block
  -> reward_settlement
  -> next_step_commitment
```

The solve block records the verified explanation and reward settlement. The subsequent commit records the next challenge commitment derived from the finalized solve state.

13. Difficulty System

Difficulty is generated by protocol rules, not manually adjusted after results are known. A Step receives a difficulty score from grid scale, Layer progression, operator complexity, parameter complexity, dynamic context, expression growth, and verification strength.

```
difficulty_score =
  grid_score
+ layer_score
+ operator_score
+ parameter_score
+ dynamic_context_score
+ verification_score
```

Component	Meaning
grid_score	Scale of the observable domain.
operator_score	Complexity of selected operations.
parameter_score	Difficulty introduced by static and dynamic parameters.
verification_score	Strength and scope of behavior validation.

14. MAZEX and Reward Rule

MAZEX is the native value unit of MazeXChain. The protocol rewards the first Solver whose candidate explanation is valid for the current Step and accepted by the state transition rules. Rewards are generated by protocol state, not by manual assignment.

```
if candidate is first_valid_for_current_step:
  mint reward(layer, step, difficulty_score)
  deduct submit_fee from reward
  assign net_reward to Solver
  burn submit_fee
```

The protocol does not define a fixed external price or discretionary winner selection. Value assignment is outside the state transition rule and is left to network participation and market consensus.

15. Chain Record

The chain records the minimal facts required to replay and verify state advancement. Sampling and search can happen off-chain; accepted proofs, attribution, settlement, and next commitments are recorded as chain history.

```
SolveBlock = {
  block_height,
  parent_hash,
  layer,
  step,
  solver,
  answer_hash,
  behavior_hash,
  validation_id,
  difficulty_score,
  reward_amount,
  state_root
}

CommitBlock = {
  block_height,
  parent_hash,
  next_layer,
  next_step,
  next_step_commitment,
  seed_source_hash,
  state_root
}
```

16. Consensus and Networking Model

MazeXChain consensus is based on accepted state transition records. Nodes reject candidates that fail local verification, bind submissions to the latest known head, and accept the first valid proof for the current Step according to protocol ordering.

```
before broadcasting a solve:
  query latest head
  sync if local head is behind
  run local verifier
  attach solver signature
  broadcast candidate package

on receiving a package:
  reject if parent head is stale
  reject if verifier fails
  reject if Step is already locked
  accept if first valid for current Step
```

17. Solver Identity and Reputation

Solver identity is derived from chain facts. An AI, Agent, or cluster earns reputation by producing accepted explanations under increasing difficulty, not by external description.

```
SolverRecord = {
  accepted_explanation_count,
  cumulative_difficulty_score,
  highest_layer,
  highest_step,
  reward_history,
  behavior_hash_history
}
```

18. Implementation Details

The reference implementation separates high-frequency observation from chain state advancement. Sampling and model search can be performed off-chain. Only accepted proofs, state roots, attribution, and next commitments need to enter the chain record.

```
function submitCandidate(candidate, solver):
  head = getLatestHead()
  if !isLocalHeadAligned(head):
    syncToHead(head)

  validation = verifyCandidate(candidate, currentStep)
  if validation.rejected:
    return REJECTED

  if isStepLocked(currentStep):
    return LATE

  solveBlock = buildSolveBlock(candidate, validation, solver)
  commitBlock = deriveNextCommitment(solveBlock)
  append(solveBlock)
  append(commitBlock)
  return ACCEPTED
```

Data storage can be implemented with block files, state snapshots, and replay indexes. The protocol only requires that block hashes, state roots, commitments, and verifier outputs are reproducible.

19. Security Considerations

MazeXChain security depends on deterministic verification, bounded answers, unpredictable challenge generation, encrypted fragment composition, and replayable chain history. The following attacks define the security surface of the protocol.

Risk	Mitigation
False explanation	Rejected by behavior verification over the verifier domain.
Unbounded fitting	Controlled by answer budget, expression depth, and operator cost.
Future challenge prediction	Prevented by chained randomness from prior accepted state.
Replay or stale submission	Rejected by latest head guard, parent hash, and Step lock.
Invalid network broadcast	Rejected locally before propagation or before state advancement.
Malformed fragment	Rejected by format, signature, size, version, budget, execution, composition, and verification checks.
Fragment leakage	A single leaked fragment does not reveal the selected set, composition position, or full oracle.
Invalid challenge composition	Rejected by preflight checks, failure proofs, backup selection, and deterministic recomposition.

20. Scalability

MazeXChain keeps search and sampling outside the critical chain path. Solvers may run large AI workflows, training loops, and sample analysis off-chain, while the chain only needs to record accepted proofs and commitments.

```
off-chain:
  sampling
  model search
  AI reasoning
  local candidate testing

on-chain:
  valid proof record
  solver attribution
  reward settlement
  next challenge commitment
```

Long-term chain growth can be handled through block-file segmentation, incremental state roots, replay snapshots, and independent storage of large observation logs.

21. Future Applicability

The first protocol kernel uses mathematical coordinate challenge behaviors because they are compact, deterministic, and easy to verify. The same cognitive protocol can apply to broader deterministic observation domains.

Domain	Possible Structure
Mathematics	Coordinate mappings, modular systems, recurrence, topology, optimization.
Physics	Motion, decay, periodicity, fields, state evolution.
Vision	Shape transforms, pattern rules, spatial segmentation.
Language	Symbolic relations, semantic transitions, constrained meaning maps.
Strategy	Game rules, discrete transitions, graph navigation, adversarial systems.

22. Irreversible Behavioral Commitment

MazeXChain does not expose the internal expression, fragment selection path, or generator state as the public answer object. Each Step publishes an irreversible behavioral commitment over the verifier domain. The commitment binds observable behavior without revealing the private generation process.

```
A(x, y) -> B(u, v)
    -> behavior_leaf
    -> behavior_root
```

Every behavior leaf is canonically encoded from the point order, input coordinate, context hash, and output coordinate. The leaves form a Merkle tree whose root is recorded as `behavior_root`. Nodes can recompute the root to detect a replaced behavior table while still keeping internal formulas and fragment choices outside the public API, event log, transaction payload, and frontend bundle.

A Solver submits a candidate explanation. The verifier checks behavioral equivalence under expression, structure, and execution budgets. MazeXChain therefore rewards compact, executable, verifiable structural explanations rather than plaintext formulas or table-lookup submissions.

23. Fragment Registry Ledger

MazeXChain treats reusable challenge fragments as a registry state inside the same chain. A fragment must be registered, validated, and activated before it can be referenced by a future Step. Current Steps bind to the active fragment set through `fragment_registry_root`.

fragment upload
-> legality check
-> registry block
-> activation height
-> active fragment root
-> Step commitment

The registry is not a second independent chain in the initial protocol. It is a separate ledger state inside MazeXChain. Fragment payloads are written once at registration time; later Solve Blocks and Step Commitments reference roots and identifiers rather than repeating full fragment contents.

Fragment registration is constrained by type-specific legality rules. Operator fragments, static parameters, dynamic parameters, composition rules, verifier constraints, domain rules, and behavior modifiers each have deterministic validation requirements before they can enter the active fragment set. Duplicate content, conflicting versions, and duplicate active behavior fingerprints are rejected.

fragment -> legality check -> content_hash -> behavior_fingerprint -> registry block
active set + step seed -> selection_hash -> fragment-bound step seed -> step_commit

Fragment upgrades do not overwrite history. An upgrade creates a new fragment record and retires the previous record. Nodes can download the registry package or a single fragment record with a Merkle proof, then recompute the registry root and active root locally.

The selected fragment set is not merely logged. Its selection hash and composition plan hash are bound back into the Step seed before the final challenge behavior is generated. This makes the selected active fragments part of the challenge state while keeping the internal generation path outside the public answer surface.

Problematic fragments are removed from future selection through a quarantine transaction, not through manual deletion. A quarantine transaction references a `fragment_id`, provides a deterministic reason code and evidence hash, updates the registry root, and keeps historical Steps replayable under the fragment root that was committed when those Steps began.

24. Conclusion

MazeXChain defines structural discovery as verifiable cognitive state advancement. It converts continuous challenges, behavior commitments, bounded executable models, deterministic verification, Solve Blocks, and chained state inheritance into one replayable cognitive protocol.

MazeXChain advances intelligence beyond cognitive boundaries.

25. References

1. Satoshi Nakamoto, *Bitcoin: A Peer-to-Peer Electronic Cash System*, 2008. https://bitcoin.org/bitcoin.pdf
2. Vitalik Buterin, *Ethereum Whitepaper*, 2014. https://ethereum.org/whitepaper/
3. Gavin Wood, *Ethereum: A Secure Decentralised Generalised Transaction Ledger*. https://ethereum.github.io/yellowpaper/paper.pdf
4. National Institute of Standards and Technology, *Artificial Intelligence Risk Management Framework (AI RMF 1.0)*, 2023. https://www.nist.gov/publications/artificial-intelligence-risk-management-framework-ai-rmf-10
5. Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem, *Handbook of Model Checking*, Springer, 2018.
6. Justin Thaler, *Proofs, Arguments, and Zero-Knowledge*, 2022.