

Proof Protocol: A Verifiable Execution Layer for AI-Generated and Programmatic Computation

Anonymous¹

¹Preprint — submitted to arXiv cs.CR / cs.DC

June 27, 2026

Abstract

We present PROOF PROTOCOL, a system that transforms arbitrary computational claims — including those generated by large language models (LLMs) — into verifiable, reproducible, and cryptographically signed execution artifacts we call *Computational Proof Objects* (CPOs). A CPO binds a natural-language hypothesis, executable code, a controlled sandbox specification, deterministic execution results, a SHA-256 content hash, and an Ed25519 digital signature issued by the executing node.

Our core contribution is a *replay-based verification model*: rather than constructing zero-knowledge proofs, a verifier re-executes the same code under identical constraints and compares observable outputs. This approach requires no trusted setup, no circuit compilation, and no global consensus, while providing integrity, authenticity, and bounded reproducibility under formally stated assumptions. We define the CPO data model, formalize the verification function as a decidable predicate, characterize the cost structure of proof generation versus verification, and analyze security properties with explicit non-goals.

The system supports seven isolated execution environments — *worlds* — spanning symbolic mathematics, probabilistic inference, evolutionary computation, formal verification, and multimodal processing. A reference implementation in Python (FastAPI, Ed25519 via the `cryptography` library, Docker) is publicly available.

Contents

1	Introduction	3
2	The CPO Data Model	4
3	System Architecture	4
4	Execution Model	5
4.1	Bounded Execution Environment	5
4.2	Observable Equivalence	5
5	Cryptographic Layer	6
5.1	Canonical Serialization	6
5.2	Content Hash and Signature	6
5.3	Node Identity	6

6	World Abstraction	6
7	Verification Procedure	7
7.1	Formal Verification Function	7
7.2	Algorithm	7
8	Cost Analysis	7
8.1	Proof Generation	8
8.2	Verification	8
8.3	Comparison with ZK-based Systems	8
9	Distributed Network Model	9
9.1	Network Participants	9
9.2	Versioned World Registry	10
9.3	CPO Lifecycle State Machine	10
9.4	Quorum Attestation	11
9.5	Conflict Detection and Resolution	11
10	Security Analysis	11
10.1	Threat Model	11
10.2	Non-Goals	12
11	Limitations	12
12	Related Work	13
13	Conclusion	13

1 Introduction

Modern AI systems and automated computation pipelines share a critical weakness: their outputs are difficult to verify independently. A language model may produce a plausible-sounding numerical answer that is arithmetically incorrect; a data-science pipeline may yield different results on different machines; an autonomous agent may take actions whose causal chain is impossible to reconstruct after the fact.

Three distinct communities experience this problem in parallel:

AI practitioners observe that LLM outputs can *hallucinate* facts, including mathematical results, without any signal to the consumer that the output is unreliable [Ji et al., 2023]. Grounding LLM outputs in code execution partially addresses this, but does not provide an auditable record.

Scientists struggle with *computational reproducibility*. A 2016 survey found that more than 70% of researchers had failed to reproduce another scientist’s results, with software environment differences as a major contributing factor [Baker, 2016].

Auditors and compliance engineers require *auditability*: the ability to prove *post hoc* that a particular output was produced by a particular computation running in a particular environment.

Existing solutions address each concern in isolation. Reproducibility tools (MLflow [Zaharia et al., 2018], DVC [Rumelhart et al., 2020]) log hyperparameters and artifacts but do not cryptographically bind execution environments to outputs. Containerization (Docker, Nix [Dolstra, 2006]) improves environment reproducibility but does not attest to execution results. Verifiable computation systems (Truebit [Deutsch & Reitwießner, 2019], RISC Zero [RISC Zero, 2022]) provide unconditional soundness via zero-knowledge proofs, but impose circuit compilation overhead that is orders of magnitude larger than the computation itself — making them impractical for lightweight interactive use.

PROOF PROTOCOL occupies a different position in this design space: a *practical, lightweight verifiable execution layer* that cryptographically binds computation inputs to outputs using replay-based verification rather than zero-knowledge machinery. The key insight is that *reproducible isolation is a sufficient substitute for cryptographic universality* in many real-world settings: if the execution environment is tightly controlled, re-execution is both cheap and conclusive.

Contributions

1. A formal data model for Computational Proof Objects (CPOs), including a definition of their content hash and node signature (Section 2).
2. A formal, decidable verification function `Verify` with explicit assumptions and cost characterization (Sections 7, 8).
3. A formal definition of *observable equivalence* that precisely specifies when two executions are considered equal (Section 4.2).
4. A distributed network model formalising quorum attestation, world versioning, CPO lifecycle states, and conflict resolution (Section 9).
5. A security analysis with an explicit threat model and non-goals (Section 10).
6. A reference implementation demonstrating practical deployability.

2 The CPO Data Model

Definition 1 (Computational Proof Object). A Computational Proof Object (CPO) is an immutable tuple

$$\mathcal{P} = (id, w, claim, code, env, R, h, \sigma, pk, t)$$

where:

- $id \in \{0, 1\}^{128}$ is a globally unique identifier (UUIDv4), sampled uniformly at proof-generation time.
- $w \in \mathcal{W}$ selects an execution environment from a finite, registered set of worlds (Section 6).
- $claim \in \Sigma^*$ is a natural-language description of the hypothesis being tested.
- $code \in \Sigma^*$ is the executable source code that operationalizes the claim.
- $env = (image, C)$ is an execution specification, where *image* is an optional container image override and *C* is a constraint map (memory limit, timeout, etc.).
- $R = (s_{out}, s_{err}, e, \tau)$ is the observed execution result: stdout, stderr, exit code, and wall-clock runtime in milliseconds.
- $h \in \{0, 1\}^{256}$ is the content hash (Definition 2).
- $\sigma \in \{0, 1\}^{512}$ is an Ed25519 signature over h produced by the node’s private key sk .
- $pk \in \{0, 1\}^{256}$ is the Ed25519 public key of the signing node.
- $t \in \mathbb{Z}$ is the UTC Unix timestamp of creation.

Definition 2 (Content Hash). Let $\mathcal{P}^- = \mathcal{P} \setminus \{h, \sigma\}$ denote the CPO with the mutable fields removed. The content hash is:

$$h(\mathcal{P}) = \text{SHA-256}(\text{canon}(\mathcal{P}^-)) \tag{1}$$

where *canon* is a deterministic serialization function (Section 5.1).

The hash h covers every semantically meaningful field, so tampering with any single field produces a different hash that invalidates the stored signature.

3 System Architecture

Figure 1 illustrates the end-to-end pipeline.

API surface. The system exposes four endpoints:

- `POST /prove` — accept $(w, claim, code)$, execute in sandbox, produce and store \mathcal{P} ; return h and σ .
- `POST /ask` — accept a natural-language question q , generate *code* via an LLM, then delegate to `/prove`.
- `GET /verify/{h}` — retrieve \mathcal{P} by h , re-execute, return verification verdict.
- `GET /ledger` — browse the append-only log, optionally filtered by world w .

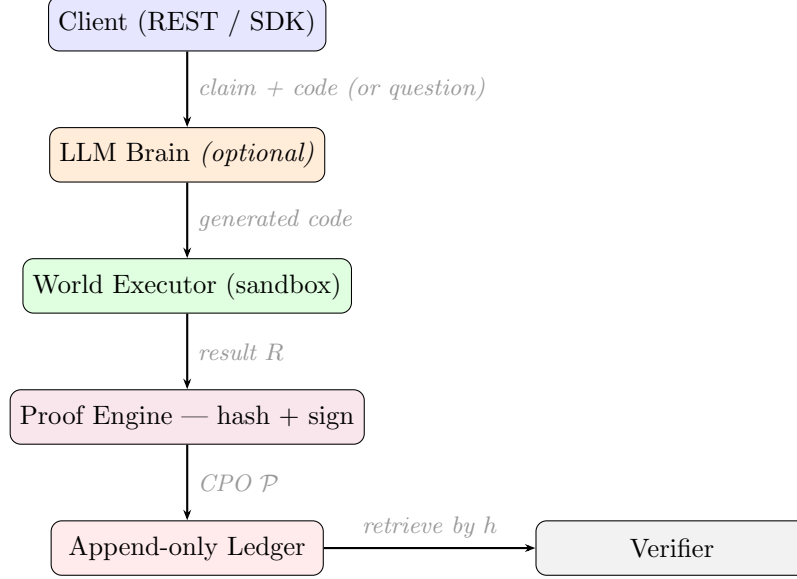


Figure 1: Proof Protocol pipeline. The LLM brain is activated only by `POST /ask`; direct code submission uses `POST /prove`. Any stored CPO can be retrieved and re-verified offline via `GET /verify/{hash}`.

4 Execution Model

4.1 Bounded Execution Environment

Each execution runs inside an ephemeral container with the following constraints enforced at runtime:

- **Network isolation:** all inbound and outbound traffic is suppressed (`network_disabled=True`).
- **Read-only filesystem:** no persistent side effects (`read_only=True`).
- **Memory cap:** default 128 MiB; configurable via `env.C`.
- **Execution timeout:** default $\delta = 10$ s wall-clock.
- **Privilege suppression:** `no-new-privileges:true` blocks `setuid`-based escalation.

We call this combination a *bounded execution environment* (BEE) and denote the constraints as $\text{BEE}(w, C)$.

4.2 Observable Equivalence

We define precisely when two execution results are considered equal during verification.

Definition 3 (Observable Equivalence). *Let $R = (s_{\text{out}}, s_{\text{err}}, e, \tau)$ and $R' = (s'_{\text{out}}, s'_{\text{err}}, e', \tau')$ be two execution results. We say $R \approx R'$ (observably equivalent) if and only if:*

$$\text{norm}(s_{\text{out}}) = \text{norm}(s'_{\text{out}}) \quad (2)$$

$$e = e' \quad (3)$$

where $\text{norm}(s) = s.\text{strip}()$ removes leading and trailing whitespace and normalizes line endings to $\backslash n$.

The stderr stream and runtime τ are deliberately excluded from \approx : stderr content can vary with library version, and runtime is non-deterministic by nature. Only the semantic output (stdout) and success indicator (exit code) are equality-tested.

Definition 4 (Bounded Determinism). *A code/environment pair $(\text{code}, \text{BEE}(w, C))$ is boundedly deterministic if for all executions E_1, E_2 producing results R_1, R_2 :*

$$R_1 \approx R_2$$

Bounded determinism is a *sufficient*, not necessary, condition for successful re-verification. Stochastic programs can still be verified for integrity and authenticity (steps 1–2 of Algorithm 1) even when output equality (steps 3–4) does not hold.

5 Cryptographic Layer

5.1 Canonical Serialization

To ensure h is stable across serialization implementations, we compute canon using an RFC 8785-inspired deterministic JSON form: keys are sorted lexicographically at every nesting level, separators carry no extra whitespace, and Unicode code points are preserved without escaping:

```
canon(O) = JSON.dumps(O, sort_keys=True, separators=(",", ":"), ensure_ascii=False)
```

This ensures that two identical CPOs serialized independently produce bitwise-identical byte strings before hashing.

5.2 Content Hash and Signature

The content hash (Equation (1)) is computed over the UTF-8 encoding of $\text{canon}(\mathcal{P}^-)$. The node signs the hex-encoded hash string:

$$\sigma = \text{Ed25519.Sign}(sk, \text{hex}(h) \cdot \text{UTF-8})$$

Property 1 (Tamper Evidence). *For any CPO \mathcal{P} and any field $f \in \mathcal{P}^-$, the modified tuple $\mathcal{P}[f \mapsto f']$ satisfies $h(\mathcal{P}[f \mapsto f']) \neq h(\mathcal{P})$ with probability $1 - 2^{-256}$ (SHA-256 collision resistance), which implies $\text{Ed25519.Verify}(pk, h', \sigma) = \perp$.*

5.3 Node Identity

Each node generates an Ed25519 keypair at first startup. The node identifier is a human-readable prefix:

$$\text{node_id} = \text{hex}(\text{SHA-256}(pk))[:16]$$

The full public key pk is stored inline in every CPO, so verification requires no external key registry.

6 World Abstraction

A *world* $w \in \mathcal{W}$ is a named, versioned OCI container image providing a specific runtime with its library ecosystem and behavioral assumptions.

Table 1: Registered worlds ($|\mathcal{W}| = 7$) and their runtime environments. Worlds marked *bounded* are stochastic by design; re-verification confirms integrity but not output equality without explicit seed fixation.

World	Primary domain	Key libraries	Output equality
llm	General Python	stdlib	exact
symbolic	Symbolic mathematics	SymPy, Z3	exact
formal	Logic / SMT solving	Z3, py-aiger	exact
bayesian	Probabilistic inference	NumPyro, PyMC	bounded
evolutionary	Agent-based simulation	DEAP, Mesa	bounded
multimodal	Vision / audio	PyTorch CPU, Pillow	exact
neuro	Neural pipeline tooling	DSPy, LangChain	bounded

7 Verification Procedure

7.1 Formal Verification Function

Definition 5 (Verification Predicate). *Let \mathcal{P} be a CPO retrieved from the ledger by content hash h . Define the boolean verification predicate:*

$$\text{Verify}(\mathcal{P}) = \underbrace{\text{SigOK}(\mathcal{P})}_{\text{authenticity}} \wedge \underbrace{\text{HashOK}(\mathcal{P})}_{\text{integrity}} \wedge \underbrace{\text{ReplayOK}(\mathcal{P})}_{\text{reproducibility}} \quad (4)$$

where:

$$\text{SigOK}(\mathcal{P}) \triangleq \text{Ed25519.Verify}(\mathcal{P}.pk, \mathcal{P}.h, \mathcal{P}.\sigma) \quad (5)$$

$$\text{HashOK}(\mathcal{P}) \triangleq h(\mathcal{P}) = \mathcal{P}.h \quad (6)$$

$$\text{ReplayOK}(\mathcal{P}) \triangleq \text{Exec}(\mathcal{P}.code, \text{BEE}(\mathcal{P}.w, \mathcal{P}.env.C)) \approx \mathcal{P}.R \quad (7)$$

and Exec denotes a fresh sandbox execution returning result R' .

The three sub-predicates are independent and provide layered guarantees: SigOK and HashOK require only the stored CPO (no re-execution); ReplayOK requires re-execution and holds only under bounded determinism (Definition 4).

7.2 Algorithm

Algorithm 1 gives the procedural form used in the implementation. Figure 2 visualizes the decision flow.

Proposition 1 (Verify is Decidable). *Algorithm 1 terminates in finite time for any input \mathcal{P} , because: (i) SHA-256 and Ed25519.Verify run in $O(|\text{canon}(\mathcal{P}^-)|)$ time; (ii) Exec is bounded by the timeout $\delta \in \mathcal{P}.env.C$; (iii) norm and equality checks are $O(|s_{\text{out}}|)$.*

8 Cost Analysis

A key advantage of replay-based verification over ZK-based systems is its *symmetric cost structure*: verification costs approximately the same as proof generation, rather than imposing a proving overhead.

Algorithm 1 $\text{Verify}(\mathcal{P})$

Require: CPO \mathcal{P} retrieved from ledger by h

Ensure: Boolean verdict $v \in \{\top, \perp\}$, reason string

```
1:  $h' \leftarrow \text{SHA-256}(\text{canon}(\mathcal{P}^-))$ 
2: if  $h' \neq \mathcal{P}.h$  then return  $(\perp, \text{"hash mismatch"})$ 
3: end if
4: if  $\neg \text{Ed25519.Verify}(\mathcal{P}.pk, h', \mathcal{P}.\sigma)$  then return  $(\perp, \text{"invalid signature"})$ 
5: end if
6:  $R' \leftarrow \text{Exec}(\mathcal{P}.code, \text{BEE}(\mathcal{P}.w, \mathcal{P}.env.C))$ 
7: if  $\text{norm}(R'.stdout) \neq \text{norm}(\mathcal{P}.R.stdout)$  then return  $(\perp, \text{"stdout mismatch"})$ 
8: end if
9: if  $R'.exit\_code \neq \mathcal{P}.R.exit\_code$  then return  $(\perp, \text{"exit code mismatch"})$ 
10: end if return  $(\top, \text{"verified"})$ 
```

8.1 Proof Generation

Let T_{exec} be the sandbox execution time for code of input size n . Proof generation incurs:

1. **Sandbox execution:** $T_{\text{exec}}(n)$ — the dominant term.
2. **Canonical serialization:** $O(|\mathcal{P}|)$ — proportional to CPO size, typically $O(n + |s_{\text{out}}|)$.
3. **SHA-256 hash:** $O(|\text{canon}(\mathcal{P}^-)|)$ — hardware-accelerated on modern CPUs; negligible in practice.
4. **Ed25519 sign:** $O(1)$ — constant time, $\approx 50\mu\text{s}$ on commodity hardware.

Total: $T_{\text{prove}} = T_{\text{exec}}(n) + O(|\mathcal{P}|)$.

8.2 Verification

Verification adds one re-execution to steps (2)–(4) above:

$$T_{\text{verify}} = T'_{\text{exec}}(n) + O(|\mathcal{P}|) + O(|s_{\text{out}}|)$$

where T'_{exec} is the re-execution time. Under bounded determinism, $T'_{\text{exec}} \approx T_{\text{exec}}$. The cryptographic overhead is $O(|\mathcal{P}|)$ and empirically sub-millisecond.

8.3 Comparison with ZK-based Systems

In ZK-based systems (e.g., RISC Zero), proving time is $T_{\text{ZK}} \approx \alpha \cdot T_{\text{exec}}$ where $\alpha \gg 1$ (commonly 10^3 – $10^6 \times$ depending on circuit complexity), and verification is $O(|proof|)$ — much cheaper. PROOF PROTOCOL reverses this asymmetry:

Table 2: Cost structure comparison.

System	Proof generation	Verification
PROOF PROTOCOL	$T_{\text{exec}} + O(\mathcal{P})$	$T_{\text{exec}} + O(\mathcal{P})$
ZK (e.g., RISC Zero)	$\alpha \cdot T_{\text{exec}}, \alpha \gg 1$	$O(proof)$

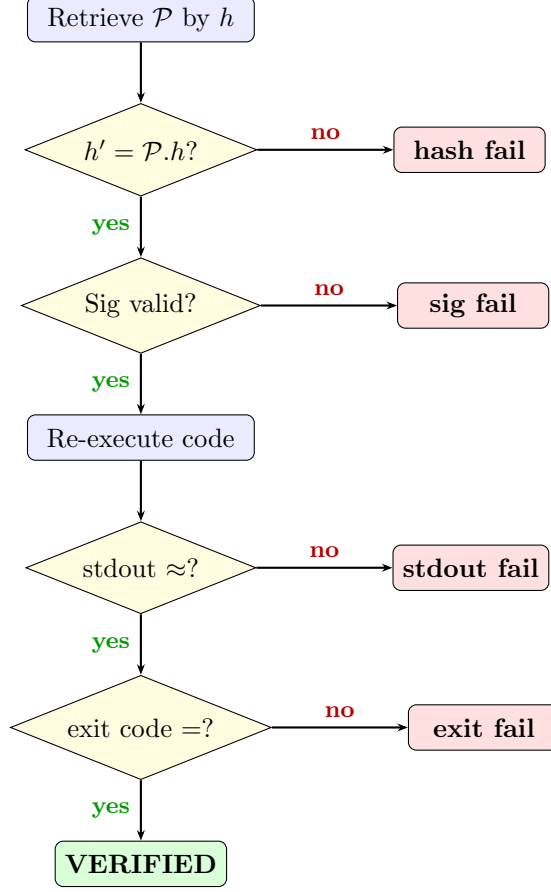


Figure 2: Decision flow of Verify (Algorithm 1). Steps 1–2 test integrity and authenticity without re-execution. Steps 3–4 test reproducibility and require a fresh sandbox run.

Replay-based verification is preferable when re-execution is cheap, the verifier controls or trusts the execution environment, and ZK overhead is prohibitive. ZK systems are preferable when re-execution is impossible or the verifier has no access to a trusted runtime.

9 Distributed Network Model

A single-node deployment provides integrity and authenticity but not independence of verification. This section formalises the distributed extension of PROOF PROTOCOL: a network of independent nodes that jointly attest CPOs via a quorum protocol, with explicit handling of world versioning and output divergence.

9.1 Network Participants

Definition 6 (Proof Network). *A Proof Network is a tuple $\mathcal{G} = (\mathcal{N}, \mathcal{W}_v, \mathcal{L})$ where:*

- $\mathcal{N} = \{n_1, \dots, n_m\}$ is a finite set of nodes, each holding a unique Ed25519 keypair (sk_i, pk_i) .
- \mathcal{W}_v is a versioned world registry (Definition 7).
- \mathcal{L} is a shared append-only ledger, replicated across nodes via gossip.

9.2 Versioned World Registry

A world identifier alone is insufficient for cross-node reproducibility: two nodes running `symbolic:latest` may use different image layers if one has not pulled recently. We require pinned image digests.

Definition 7 (Versioned World Registry). A versioned world registry \mathcal{W}_v is a mapping

$$\mathcal{W}_v : \text{world_name} \longrightarrow (\text{image_tag}, \text{digest}, \text{valid_from})^*$$

where $\text{digest} \in \{0, 1\}^{256}$ is the SHA-256 content hash of the OCI image manifest, and $\text{valid_from} \in \mathbb{Z}$ is the Unix timestamp from which this version is considered canonical.

A CPO \mathcal{P} is world-version-consistent with \mathcal{W}_v if the image digest stored in $\mathcal{P}.\text{env}.\text{image_digest}$ matches the active entry of $\mathcal{W}_v(\mathcal{P}.w)$ at time $\mathcal{P}.t$.

Requiring the image digest in env (and thus in h) ensures that world-version drift cannot produce a silent $\text{ReplayOK} = \perp$: a version mismatch is explicit and attributable.

9.3 CPO Lifecycle State Machine

Each CPO progresses through a finite set of network-observable states.

Definition 8 (CPO State). Let $\mathcal{S} = \{\text{PROPOSED}, \text{ATTESTED}, \text{CONTESTED}, \text{VERIFIED}, \text{INVALID}\}$ be the set of CPO network states. A CPO is born in **PROPOSED** and transitions according to the rules of Table 3.

Table 3: CPO lifecycle transitions. $A(\mathcal{P})$ denotes the set of nodes that have broadcast an attestation for \mathcal{P} .

From	Condition	To	Action
PROPOSED	$ A(\mathcal{P}) \geq k$, no conflict	ATTESTED	broadcast attestation
PROPOSED	conflict detected (Def. 10)	CONTESTED	broadcast conflict notice
ATTESTED	$\text{Verify}(\mathcal{P}) = \top$ by k nodes	VERIFIED	mark ledger entry
ATTESTED	conflict detected post-attest	CONTESTED	broadcast conflict notice
CONTESTED	conflict resolved (Def. 11)	VERIFIED or INVALID	per resolution rule
INVALID	—	INVALID	absorbing state

Figure 3 visualises these transitions.

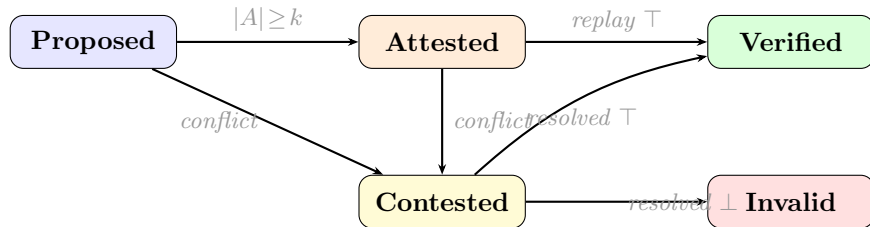


Figure 3: CPO lifecycle state machine in a distributed Proof Network. **VERIFIED** and **INVALID** are absorbing states.

9.4 Quorum Attestation

Definition 9 (Quorum). *Given a network \mathcal{G} with $|\mathcal{N}| = m$ nodes, a quorum threshold $k \in \mathbb{N}$ with $\lceil m/2 \rceil \leq k \leq m$, and a CPO \mathcal{P} , the quorum attestation predicate is:*

$$\text{Quorum}_k(\mathcal{P}) \triangleq |\{n_i \in \mathcal{N} \mid \text{Verify}_{n_i}(\mathcal{P}) = \top\}| \geq k$$

where Verify_{n_i} denotes execution of Algorithm 1 by node n_i in its own BEE.

Requiring $k > m/2$ ensures that no two disjoint quorums can attest conflicting CPOs simultaneously — a *Byzantine fault tolerance* property under the assumption that at most $\lfloor (k-1)/2 \rfloor$ nodes are compromised.

9.5 Conflict Detection and Resolution

Two nodes may independently produce CPOs for the same logical computation yet report different results — due to stochastic code, world-version drift, or a compromised node.

Definition 10 (CPO Conflict). *Two CPOs \mathcal{P} and \mathcal{P}' are in conflict if:*

$$\mathcal{P}.code = \mathcal{P}'.code \tag{8}$$

$$\mathcal{P}.w = \mathcal{P}'.w \tag{9}$$

$$\mathcal{P}.env.image_digest = \mathcal{P}'.env.image_digest \tag{10}$$

$$\mathcal{P}.R \not\approx \mathcal{P}'.R \tag{11}$$

Conditions (8)–(10) ensure the same code ran in the same world version. Condition (11) asserts that the observable outputs are not equivalent (Definition 3).

Note that if world-version-consistency (Definition 7) is enforced, most conflicts indicate either a stochastic program or a compromised node, not benign version drift.

Definition 11 (Conflict Resolution). *Given conflicting CPOs \mathcal{P} and \mathcal{P}' , the network applies the following resolution strategy in order of precedence:*

1. **Quorum rule:** if $\text{Quorum}_k(\mathcal{P})$ holds but not $\text{Quorum}_k(\mathcal{P}')$, then \mathcal{P} is VERIFIED and \mathcal{P}' is INVALID.
2. **Seed-fixation rule:** if $\mathcal{P}.w$ is a stochastic world and neither CPO fixes an explicit random seed, both are marked CONTESTED with annotation **non-deterministic**. Neither is INVALID — the conflict is informational.
3. **Attribution rule:** if neither quorum is reached for either CPO, both are marked CONTESTED and the conflicting node identities are logged for human review.

This three-rule hierarchy ensures that deterministic computations converge to a unique VERIFIED state, while stochastic ones are flagged rather than falsely invalidated.

10 Security Analysis

10.1 Threat Model

We analyze three adversary classes:

Passive ledger reader. An adversary who reads CPOs from the ledger but cannot modify them. The append-only storage structure and Property 1 ensure that any modification is detectable.

Malicious code submitter. An adversary who submits code intended to escape the sandbox or cause persistent side effects on the host. The BEE constraints (network isolation, read-only FS, memory cap, no-new-privileges) collectively reduce the surface area. We note that Docker-based isolation does not defend against kernel-level exploits (see Section 11).

Compromised node. An adversary who controls a node’s private key sk . This adversary can forge signatures for any claim, but every CPO carries pk inline, so attribution is possible. Key rotation invalidates future attestations; past CPOs remain verifiable against the original pk .

10.2 Non-Goals

PROOF PROTOCOL explicitly does not provide:

- **Semantic correctness:** $\text{Verify}(\mathcal{P}) = \top$ proves that *code* ran and produced *stdout*. It does not prove that *stdout* correctly answers *claim*.
- **Universal soundness:** unlike ZK proofs, a successful replay depends on environmental determinism, which is assumed not proved.
- **Global consensus:** no mechanism prevents two nodes from issuing conflicting CPOs for the same claim.
- **Confidentiality:** CPOs store *code* and *stdout* in plaintext.
- **Kernel isolation:** Docker shares the host kernel.

11 Limitations

Determinism scope. Stochastic worlds (Bayesian, Evolutionary, Neuro) cannot guarantee $R \approx R'$ without explicit random seed fixation in the submitted code. Submitters should set seeds to obtain full $\text{Verify} = \top$; otherwise, only $\text{SigOK} \wedge \text{HashOK}$ is guaranteed.

Container security boundary. Docker-based isolation does not protect against kernel exploits. A hardened deployment should use microVM isolation (Firecracker [Agache et al., 2020]) or a syscall-intercepting runtime (gVisor [Lacasse et al., 2020]).

LLM code quality. The `/ask` endpoint relies on an LLM to generate *correct* code. A hallucinated or buggy code snippet produces a valid, signed CPO for a wrong answer: $\text{Verify} = \top$ with incorrect semantics. Users should treat CPOs as execution certificates, not semantic truth certificates.

Ledger scalability. The current JSONL ledger is append-only and single-file. At scale, partitioning by world or time window, or migration to a structured event store (Kafka, RocksDB), is required.

Single-node trust. The reference implementation is single-node. The quorum and conflict resolution model is formalised in Section 9 but not yet implemented; a Byzantine-tolerant gossip layer is future work.

12 Related Work

Reproducible research and ML tracking. MLflow [Zaharia et al., 2018] and DVC [Rumelhart et al., 2020] record experiment parameters and artifacts but do not cryptographically bind execution environments to outputs. Nix [Dolstra, 2006] and Guix provide reproducible builds at the software delivery level, not at the runtime execution level.

Verifiable computation. Truebit [Deutsch & Reitwießner, 2019] introduces interactive dispute resolution for on-chain verification of off-chain computation. RISC Zero [RISC Zero, 2022] and StarkWare compile arbitrary programs to ZK circuits, providing computational soundness at the cost of 10^3 – $10^6\times$ proving overhead. Cartesi [Machado et al., 2021] runs Linux in a reproducible RISC-V machine, generating verifiable traces. PROOF PROTOCOL trades unconditional soundness for practical deployability: re-execution is sufficient when the verifier controls or trusts the runtime.

Trusted Execution Environments. Intel SGX and AMD SEV provide hardware attestation of code running in enclaves [Costan & Devadas, 2016], offering strong isolation without network trust assumptions. These require specific hardware and attestation infrastructure; PROOF PROTOCOL is hardware-agnostic.

Secure sandboxing. Firecracker [Agache et al., 2020] provides microVM isolation with sub-125 ms boot times, used in AWS Lambda. gVisor [Lacasse et al., 2020] intercepts syscalls in user space. Both are compatible with our execution layer and represent a natural isolation upgrade path.

LLM grounding via tool use. Toolformer [Schick et al., 2023] and ReAct [Yao et al., 2023] ground LLM outputs in external tool calls to reduce hallucination. The `/ask` endpoint instantiates this paradigm with a verifiable execution backend: the LLM proposes code, the sandbox executes it, and the CPO attests to the result.

Execution provenance. PROV-DM [Lebo et al., 2013] and related W3C provenance standards model provenance graphs for data pipelines. PROOF PROTOCOL can be viewed as a lightweight cryptographic specialization of execution provenance for sandboxed programs.

13 Conclusion

We have presented PROOF PROTOCOL, a lightweight verifiable execution layer that transforms computational claims into signed, auditable, and reproducible artifacts. The system rests on three principles:

1. **Isolation implies reproducibility.** A bounded execution environment makes replay-based verification tractable without ZK machinery.
2. **Canonical hashing implies integrity.** Any tamper with a CPO is detectable by recomputing h (Property 1).
3. **Signed attestation implies non-repudiation.** Node identity is cryptographically bound to every CPO it issues.

We formalized the CPO data model, defined observable equivalence precisely (Definition 3), expressed the verification function as a decidable predicate (Definition 5 and Algorithm 1), and characterized the symmetric cost structure that distinguishes replay-based from ZK-based verification. These properties jointly position PROOF PROTOCOL as a practical trust layer for AI pipelines, scientific computation, autonomous agents, and auditable decision systems.

Availability. The reference implementation (Python, FastAPI, Ed25519, Docker) is available at <https://github.com/haynbroit-alt/CPO>.

References

- Baker, M. (2016). 1,500 scientists lift the lid on reproducibility. *Nature*, 533(7604):452–454.
- Machado, D., et al. (2021). Cartesi: Bringing real-world computations to the blockchain. *arXiv preprint arXiv:2109.04316*.
- Agache, A., et al. (2020). Firecracker: Lightweight virtualization for serverless applications. In *17th USENIX NSDI*, pp. 419–434.
- Costan, V., and Devadas, S. (2016). Intel SGX explained. *IACR ePrint Archive*, 2016:086.
- Ji, Z., et al. (2023). Survey of hallucination in natural language generation. *ACM Computing Surveys*, 55(12):1–38.
- Zaharia, M., et al. (2018). Accelerating the machine learning lifecycle with MLflow. *IEEE Data Engineering Bulletin*, 41(4):39–45.
- Ruslan, K., et al. (2020). DVC: Data version control — git for data and models. <https://dvc.org>.
- Dolstra, E. (2006). *The Purely Functional Software Deployment Model*. PhD thesis, Utrecht University.
- Lacasse, N., et al. (2020). gVisor: An application kernel for containers. *USENIX HotCloud*.
- Teutsch, J., and Reitwießner, C. (2019). A scalable verification solution for blockchains. *arXiv preprint arXiv:1908.04756*.
- Schick, T., et al. (2023). Toolformer: Language models can teach themselves to use tools. In *Advances in NeurIPS 36*.
- Yao, S., et al. (2023). ReAct: Synergizing reasoning and acting in language models. In *ICLR 2023*.
- RISC Zero. (2022). RISC Zero: A general-purpose zero-knowledge virtual machine. <https://www.risczero.com>.
- Lebo, T., et al. (2013). PROV-DM: The PROV Data Model. W3C Recommendation. <https://www.w3.org/TR/prov-dm/>.