

CPSC 416 – Distributed Systems

Lecture 24: Building a System That Tells You It's Lying

UBC Vancouver

2026-04-02

From Last Time

Turn to your neighbor. 2 minutes.

What is the difference between crash failure, Byzantine failure, and attestation failure?

For each: what can the failing node do? What can the verifier detect?

(Pairs: 2 minutes)

The Course So Far

- **L10–L14:** Quorum intersection, Paxos, Flexible Paxos
- **L17:** Byzantine fault tolerance — what happens when nodes lie
- **L18:** The impossibility zoo — structural limits on verification
- **L22:** Who decides what to verify? (political choice in system design)

Today: what happens when the code author is the Byzantine node?

Failure-First Question

AI writes production code alongside humans.

How do you know what it wrote?

How do you know the tests it wrote actually test what they claim?

This is not hypothetical. It is happening now.

Supply Chain Attacks: Code That Looked Legitimate

SolarWinds (2020):

- Build system compromised — malicious code injected into signed updates
- 18,000 organizations installed the update
- The signature was valid. The build process was compromised.

XZ Utils (2024):

- Maintainer account socially engineered over two years
- Backdoor inserted into compression library used by `sshd`
- Discovered by accident: a developer noticed 500ms latency increase

Both attacks passed code review. Both were signed. Both were “legitimate.”

The AI Dimension

AI-generated code introduces a new variant of the same problem:

- AI can write code that passes tests — because it wrote the tests too
- AI can write plausible commit messages — because it is a language model
- AI can produce code that looks correct and does something else entirely
- AI output is not malicious (no intent) — but it can be *wrong*

The failure model:

Not Byzantine (no adversarial intent). Not crash-stop (it keeps producing output).

Something in between: a process that is confident, fluent, and potentially incorrect.

Checkpoint 1

Articulate:

What is the difference between:

- ① A node that crashes (silent)
- ② A node that lies (Byzantine)
- ③ A node that is wrong but cooperative (AI)

Which is hardest to detect? Why?

(Pairs: 2 minutes)

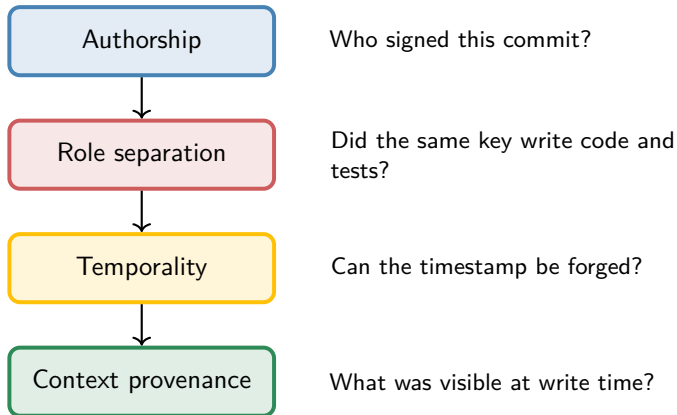
The Verification Wish List

If you wanted to trust code in a repository, you would want to verify:

- ① **Authorship:** Who wrote each change? (human vs AI vs which AI)
- ② **Role separation:** Were tests written by a different entity than the code?
- ③ **Temporality:** When was each change made? (non-forgable)
- ④ **Context provenance:** What could the author see when writing?

These are the four properties your design must address.

The Design Lattice Connection



Each property depends on the one above it.
Context provenance is meaningless without verified authorship and time.

In-Class Design Exercise

Groups of 4. You have 25 minutes.

Design an interface for a code repository that makes verifiable:

- ① Who authored each change (human vs AI vs which AI)
- ② Whether tests were written by a different entity than the code they test
- ③ When each change was made (non-forgable timestamp)
- ④ What the author could see at the time of writing (context provenance)

Deliverable: A diagram on the whiteboard. Be specific about:

- What cryptographic primitives you use
- What metadata attaches to each commit
- What the CI/CD pipeline enforces

Design Constraints

Your design must address:

- The AI has access to the same tools as the human
- The AI can generate any text, including commit messages and test names
- The repository is public (anyone can read it)
- You cannot modify Git's internal format
- You have access to: GPG keys, CI/CD, timestamping services, metadata fields

Hint: Think about what Paxos taught you.

The quorum intersection argument was about witnesses. Who is the witness here?

While You Work: Questions to Consider

- Can the human impersonate the AI? Can the AI impersonate the human?
- If both sign with their own key, who controls key issuance?
- What prevents backdating a commit?
- What does “context provenance” actually mean for a code change?
- Where is the trust anchor? What do you assume is not compromised?

(Groups: 25 minutes. Then present: 2 minutes each.)

Adversarial Round

Presentations done.

Now: break your neighbor's design.

For the group to your right:

- ① What can be forged?
- ② What can be backdated?
- ③ Where does the trust model assume honest participants?

(5 minutes per group. Be specific.)

Checkpoint 2

Articulate:

After the adversarial round—what survived? What broke?

Was any design able to verify all four properties without trusting the author?

(Plenary: 3 minutes)

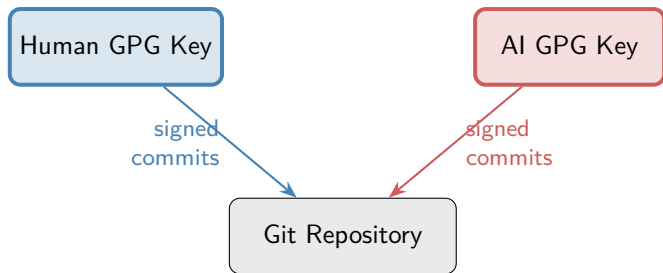
A Real Architecture

Here is one way to do it.

This is from a real system called **yanantin** — a human-AI collaborative development environment built for this exact problem.

Compare each element to what your group designed.

Element 1: Cryptographic Authorship



- Two distinct GPG keys: one held by the human, one held by the AI agent
- Every commit is signed. `git log -show-signature` reveals authorship.
- The AI *cannot* sign as the human. The human *cannot* sign as the AI.

Property 1 (authorship): verified by cryptographic signature.

Element 2: Provenance Envelope

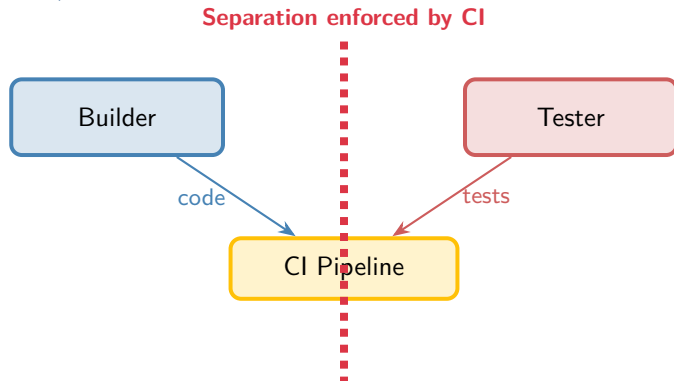
Every data record carries a **ProvenanceEnvelope**:

- **Who:** Which key signed this record
- **What:** Hash of the record content
- **When:** Timestamp (see Element 4)
- **How:** Tool chain that produced the record
- **Why:** Reference to the prompt, task, or issue that triggered it

The envelope is *metadata on every record*, not just on commits.

This is data-level lineage, not just version control lineage.

Element 3: Builder/Tester Separation



- The entity that writes code **must not** write the tests for that code
- CI enforces: if the same key signed both the source file and its test, the build fails
- This is *structural*, not policy. The pipeline rejects violations automatically.

Property 2 (role separation): enforced by CI, not by trust.

Element 4: Non-Forgeable Timestamps

OpenTimestamps:

- Hash of each commit is submitted to a timestamping authority
- The authority includes the hash in a Merkle tree
- The Merkle root is anchored in the Bitcoin blockchain
- Result: a cryptographic proof that this commit existed at this time

Why blockchain here?

Not because blockchain is a good database. Because blockchain provides a *public, append-only, non-forgeable ordering* that no single party controls.

Property 3 (temporality): anchored in a public ledger.

Recall L17: blockchain is one answer to “what if you can’t enumerate participants?”

Element 5: DecoderRing

Problem: Metadata labels like “author: AI” are meaningful to insiders. An attacker who reads the metadata knows what to forge.

Solution: **semantically meaningless labels.**

- Internal identifiers are opaque UUIDs
- The mapping from UUID to meaning lives inside the trust zone
- Outside the trust zone, the labels convey no information

This is a **defense-in-depth** measure:

Even if an attacker reads the envelope, they cannot determine *what the fields mean* without access to the DecoderRing.

Security through structure, not through obscurity.

What Yanantin Does NOT Solve

The AI can still lie in the content it signs.

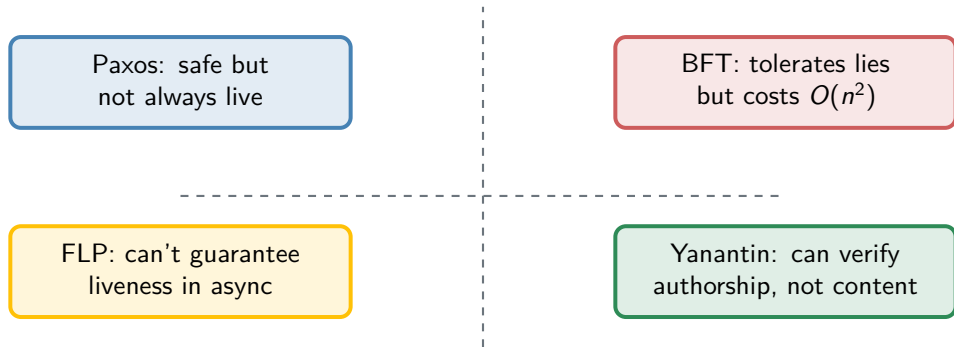
- Signatures prove authorship, not correctness
- A signed commit that says “fix: resolve null pointer” might introduce a new bug
- A signed test that says “test: verify authentication” might test nothing
- The provenance envelope tells you *who wrote it and when*
- It does not tell you *whether it is right*

This is the wall.

Verification of authorship and lineage is tractable.

Verification of *semantic correctness* is, in general, undecidable.

The Analogy to Distributed Systems



Every impossibility tells you the same thing:
Here is what you can verify. Here
is where the evidence runs out.
You must decide what to do at the boundary.

Testimony vs Telemetry

Telemetry: Data generated by a system about its own behavior.

- Logs, metrics, traces
- Machine-generated, machine-readable
- **Verifiable:** checksums, cryptographic proofs, causal ordering

Testimony: A claim made by an agent about what it did and why.

- Commit messages, PR descriptions, documentation
- Agent-generated, human-readable
- **Not independently verifiable:** you can check *that* it was said, not *that* it is true

The design question: How much of your system's evidence is testimony?
How much is telemetry?

Checkpoint 3

Articulate:

In the yanantin architecture, which of the five elements produce *telemetry* (independently verifiable) and which produce *testimony* (claims you must choose to trust)?

(Pairs: 2 minutes)

The Course Arc

L12: You can't guarantee your own liveness (FLP)



L17: Nodes can lie. Tolerance costs $3f + 1$.



L18: Every impossibility says the same thing

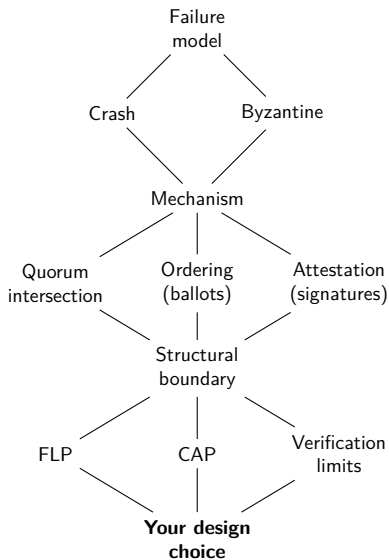


L22: Who decides what to verify?



L24: Here's what you CAN verify, and here's the wall

The Design Lattice, One Last Time



Where Does Your Capstone Sit?

Every system you build in your capstone project makes choices:

- What failure model are you assuming?
- What mechanism handles that failure?
- Where does your design hit a structural boundary?
- What do you do at the boundary?

Your final report should answer these questions explicitly.

If you can place your system on the lattice, you understand what you built.

If you cannot, that is the most important thing to figure out before you submit.

Evidence Question

You are reviewing a pull request. The commit is signed by an AI key. The tests pass. The CI pipeline shows builder/tester separation.

What can you verify about this PR?

What can you NOT verify?

What would you need to check manually?

(This is the question your capstone report should be able to answer about your own system.)

What We Learned

- Supply chain attacks and AI-generated code share a common structure: *code that looks legitimate but may not be*
- **Authorship** can be verified with cryptographic signatures
- **Role separation** can be enforced structurally in CI
- **Temporality** can be anchored in a public ledger
- **Context provenance** is the hardest — and most valuable — property
- **Semantic correctness remains unverifiable in general**
- Testimony (claims) and telemetry (evidence) are fundamentally different

The system that tells you it is lying is more trustworthy than the one that cannot.

The Course in One Sentence

Every distributed system is a set of claims
about what happened, and to whom.

The design question is always the same:

What evidence do you require before you act on a claim?

You now have the vocabulary to answer that question precisely.

Logistics

- **This is the last content lecture.**
- **L25 (Apr 7):** Course recap and open Q&A
- **Final submission:** Repository + report (see Canvas for deadlines)
- No capstone presentations — the repo and report are the deliverable
- **Self-assessment:** Opens 13:00, due Apr 7 11:30 (Pacific)

Thank you for a remarkable term.