Intelligence Economy
INSTITUTE

The Trillion-Dollar Proof: Formal Verification, the Tooling Landscape, and the Two Layers Blockchain Security Is Missing

---

Zakaryae Boudi

Working Paper
research@tokenfrastructure.com

June 3, 2026

The stakes: a multi-trillion-dollar market migrating onto rails it has not learned to secure

Tokenization is no longer a thesis; it is a migration. The on-chain real-world-asset market sits around $20–22 billion today, and the institutions modeling its trajectory are not arguing about whether it grows but by how much. Citi's *Tokenization 2030: Wall Street On-Chain* puts the base case at $5.5 trillion by 2030; JPMorgan models $13 trillion; the more bullish houses (Ark Invest, BCG/Ripple, Standard Chartered) run from $11 trillion to $30 trillion by the decade's end. These are sell-side projections and should be read with the appropriate discount; they disagree by a factor of five. But the disagreement is about magnitude, not direction. Treasuries, equities, private credit, money-market funds, and trade-finance instruments are moving onto programmable rails, and the value whose safety depends on smart-contract logic is set to climb by roughly three orders of magnitude.

Now hold that growth curve against the security record of the rails it is migrating onto. 2025 was the worst year for crypto losses on record. Hacken counted more than $3.1 billion stolen in the first half alone, already above the entire 2024 total. But the *shape* of those losses matters more than the headline, and it complicates the easy story. According to Chainalysis and to Hacken's own breakdown, the dominant attack surface in 2025 was not smart-contract code: access-control and key-management failures accounted for roughly 59% of first-half losses, while smart-contract logic bugs were about 8%. The single largest event, the $1.46 billion Bybit compromise, was an operational and signing-process attack : a manipulated signing interface against a multisig, not a flaw in contract logic. Social engineering and impersonation surged. The uncomfortable truth for anyone selling a code-correctness solution is that *most* of today's money is lost outside the code.

So we should be precise about what formal verification does and does not address, and build the argument on that precision rather than around it.

What formal verification cannot do: it cannot stop a phished signer, a stolen deployer key, or a social-engineered approval. Those are operational, organizational, and human-factor failures, and they are — for now — where the money goes.

What formal verification can do, and why it matters more each year: First, the logic-bug slice is small in 2025 precisely because logic bugs are the one class the industry has partially learned to attack with audits, fuzzers, and verification. The operational holes are larger because they are the unhardened surface, and as key management and process mature, the relative weight of logic and economic-design flaws rises. Second, the loss share understates the tail risk: a logic flaw in a contract holding tokenized Treasuries at institutional scale is not a $40 million incident, it is a systemic one, and the catastrophic tail is exactly what sampling-based testing is worst at finding. Third, and this is the part the field underweights, formal methods are not confined to "contract logic." They verify precisely the things failing today when those things are expressed as code: the access-control state machine, the multisig and key-management policy, the bridge's validation and message-authentication logic, the upgrade/governance path. A large fraction of "operational" failures are really under-specified control logic* that was never stated precisely enough to verify. Formal verification's reach is bounded, but its boundary is further out than the "it only checks math" caricature suggests.

That reframing is the honest motivation: not "verification fixes everything," but "verification is the only discipline that makes a claim about *all* behaviors of the parts we can specify, and at trillion-dollar scale, the cost of an unspecified behavior is no longer survivable."

Two structural shifts sharpen the urgency.

The attack surface is becoming asymmetric and AI-accelerated. Security leaders are explicit that 2026 changes the tempo on both sides: defenders gain machine-speed monitoring, attackers gain AI for vulnerability research, exploit development, and social engineering at scale. In 2025, AI-driven simulation was already surfacing millions of dollars of exploitable vulnerabilities, notably, that was largely a *defensive* result, AI finding bugs before attackers did, which cuts both ways. The defender's structural disadvantage has always been coverage: an attacker needs one path, a defender must hold all of them. This is exactly the gap formal verification narrows, not by out-running the attacker, but by discharging an entire property over the whole behavioral space at once, so there is no "untested path" left for either side's AI to find. FV does not win the asymmetry race; it removes a class of the racetrack.

Autonomy is opening a genuinely new surface. As tokenized finance begins delegating decisions (underwriting, compliance checks, payment orchestration) to on-chain AI agents, it wires non-deterministic, probabilistic systems into deterministic financial infrastructure that demands auditability and provable behavior. These agents can act faster than human operators and are uniquely vulnerable to manipulation of their inputs and control paths. That mismatch is being deployed ahead of the methods to contain it, and it reframes verification from a code-time activity into a runtime, behavioral one.

Put it together: trillions of dollars of core financial instruments are moving onto rails with a multi-billion-dollar annual loss rate, against adversaries amplified by AI, with autonomous agents added to the mix. Testing—sampling a finite set of paths and hoping the untested ones are fine—is structurally unequal to that threat model. Formal verification is the only discipline that reasons over *all* behaviors of the parts it can specify.

But formal verification has its own two missing layers, and together they are the quiet reason the same exploits keep recurring. The field has built remarkable machinery for *checking* artifacts (fuzzers, static analyzers, model checkers, theorem provers) and far less for two things on either side of that machinery: **agreeing on what correctness even means before checking** (the specification layer), and **composing local proofs into a system-level guarantee** (the composition layer). The first lives scattered across audit reports, blog posts, and a few experts' heads, re-derived every engagement; the second is, at scale, still an open research problem.

This piece does four things. It lays out what formal verification is and surveys the methods and tools practitioners actually reach for, including the smart-contract-native ones. It locates the two structural gaps, specification and composition, honestly, including where they remain unsolved. It connects the specification gap to the Blockchain Property Ontology (BPO), with a concrete worked example and a place in the assurance lifecycle. And it assesses where AI genuinely helps *scale* verification, and where putting AI in the loop is itself a new attack surface.

Part I : What formal verification is, precisely

Formal verification is the act of *proving or disproving* that a system satisfies a formal specification, using the methods of mathematics rather than the methods of sampling. The word "proving" marks the categorical break from testing.

Testing, however disciplined, however high its coverage, validates a *finite* set of execution paths. It can show the presence of bugs but never their absence. Formal verification inverts this: it reasons about *all* inputs that satisfy a precondition and establishes a property across the entire behavioral space at once. A passing test says "it worked on these examples." A discharged proof obligation says "it cannot fail to hold, for any input meeting these assumptions." The gap between those two sentences is the gap between empirical confidence and mathematical certainty.

That certainty is never unconditional, and pretending otherwise is the most common way verification misleads. A proof is only as good as three things beneath it:

- **The specification** may not capture what we meant. A perfect proof of the wrong property is worse than no proof, because it wears the costume of assurance.
- **The model of the environment** may not match reality. The proof assumes an honest oracle, a live chain, a bounded adversary, and reality may violate any of them.
- **The trusted computing base (TCB)** must be trusted: the prover itself, the compiler, and the hardware. In the smart-contract setting this is not abstract. Compilers have shipped code-generation bugs; a proof about *source* does not automatically hold for the *deployed bytecode* the EVM actually executes; and an upgradeable proxy can silently replace verified code with unverified code, invalidating yesterday's proof.

Verification does not eliminate trust; it *relocates and shrinks* it, replacing a sprawling, implicit trust surface with a small, explicit, auditable one. The honest statement of any result is therefore conditional : "X holds, *assuming* the oracle is honest, the chain is live, the compiler is correct, and the specification says what we meant", and a central job of any serious verification practice is to make those assumptions explicit rather than silent. This single discipline, *labeling assurance as conditional*, is the thread that connects everything below; hold onto it.

A verification effort has three parts that must be kept distinct:

1. **The system** : the artifact whose behavior we care about (a smart contract, a consensus protocol, a payment-orchestration agent, a Merkle-tree implementation).
2. **The specification** : a precise, mathematical statement of the property the system must have (no reentrancy can violate the balance invariant; total supply is conserved across every operation; every accepted deposit appears in the tree exactly once).
3. **The verification method** : the machinery connecting the two, which either produces a proof or a counterexample.

Most of the field's pain lives in part 2. Writing the specification (agreeing on *what correctness means* is harder and more error-prone than running the prover. That is the first missing layer.

Part II : The major methods

There is no single "formal verification." There is a family of methods trading off along three axes: how much automation they offer, how expressive a property they can capture, and whether they reason about a *model* of the system or the *real artifact*. The clean slogans below ("automatic but bounded," "expressive but manual") are useful first approximations, but each carries a caveat that a practitioner will insist on, so the caveats are stated, not hidden.

1. Model checking

Idea. Build a finite (or finitely abstractable) model of the system as states and transitions, then *exhaustively* explore the reachable state space to check whether a property (typically in a temporal logic such as LTL or CTL) holds everywhere. On failure it returns a concrete **counterexample trace**, which is enormously valuable for debugging.

Strengths. Strong automation and counterexamples. You describe the system and the property; the tool searches. This makes model checking the method of choice for *concurrent and distributed* designs, where bugs hide in interleavings no human would enumerate by hand.

The honest caveat. "Automatic" is real but partial. The state-space explosion problem (states grow combinatorially with concurrent components and data width) means non-trivial systems require human-supplied *abstractions* and strengthened invariants to become tractable. The history of the field is the history of fighting explosion: symbolic model checking (representing state *sets*), bounded model checking (all behaviors to depth k, reduced to a SAT/SMT query), partial-order reduction, and abstraction. The tool is push-button only after a human has done the modeling that makes it so.

Representative tools. TLA+ (https://lamport.azurewebsites.net/tla/tla.html) with the explicit-state checker **TLC** and the symbolic checker **Apalache** (https://apalache-mc.org) for high-level concurrent and distributed designs; **SPIN** (https://spinroot.com) for protocol verification; **Uppaal** (https://uppaal.org) for real-time systems as timed automata; **Alloy** (https://alloytools.org) for structural modeling over relational first-order logic with transitive closure (the feature that gives it real expressive reach) and a bounded analyzer that finds and visualizes both satisfying and falsifying instances; and the temporal-epistemic-deontic checkers **MCMAS** (https://vas.doc.ic.ac.uk/software/mcmas/) and **AJPF** (https://github.com/mcapl/mcapl) for reasoning about knowledge and norms in multi-agent systems, not just state, which is directly relevant to verifying autonomous agents.

2. Deductive verification / interactive theorem proving

Idea. Express system and specification inside a formal logic and *construct a proof* that the system meets the spec, with a proof assistant mechanically checking every inference. Unlike model checking, this handles *infinite* state spaces and unbounded data, reasoning by induction and deduction rather than enumeration.

Strengths. Maximum expressive power and the strongest guarantees : deep, quantified, infinite-domain properties beyond model checking's reach. This is the regime of landmark results: end-to-end proofs of cryptographic protocols and OS kernels.

The honest caveat. It is labor-intensive and demands rare expertise, but "fully manual" overstates it. Modern proof assistants integrate substantial automation: SMT-backed "hammer" tactics, decision procedures, and increasingly AI-guided proof search. The human still supplies strategy, key lemmas,

and induction hypotheses, and a serious effort can cost more engineer-hours than the implementation; but the trend line is toward more automation, not less.

Representative tools. Rocq (https://rocq-prover.org), formerly Coq, on the Calculus of Constructions, where the type system is the logic and a well-typed term is a proof; **Isabelle/HOL** and **HOL** (https://isabelle.in.tum.de and https://hol-theorem-prover.org), with powerful automation atop higher-order logic; and **Lean 4** (https://lean-lang.org), which doubles as a programming language and is now a focal point for machine-assisted mathematics and AI-driven proof automation. Across all of these, proof goals are explicit and updated as tactics apply, so a human can see exactly what remains.

A distinct branch: refinement-based deductive methods (the B-Method family). Worth separating out because it is the backbone of the agent-verification work in Part IV and because its industrial track record is unusual. The **B-Method** (https://www.methode-b.com/en/b-method/) and its successor **Event-B** (https://www.event-b.org) model a system as abstract state machines with an INVARIANT and OPERATIONs guarded by preconditions, then refine the abstract model toward implementation in correctness-preserving steps, discharging proof obligations at each step (tooling: **Atelier B** (https://www.atelierb.eu/en/), and the **ProB** model checker/animator (https://prob.hhu.de)). Its defining feature is correct-by-construction development rather than after-the-fact checking, and it is one of the few formal methods with a heavyweight safety-critical deployment record, and most famously the driverless Paris Métro Line 14 signaling (https://www.clearsy.com/en/the-tools/extension-of-line-14-of-the-paris-metro-over-25-years-of-reliability-thanks-to-the-b-formal-method/). Its trade-off mirrors the rest of the family: strong guarantees and a real industrial pedigree, against significant modeling effort and specialist expertise.

3. Auto-active / SMT-backed program verification

Idea. The productive middle. The developer annotates ordinary-looking code with contracts—preconditions (`requires`), postconditions (`ensures`), loop invariants (`invariant`)—and the tool compiles code-plus-annotations into verification conditions (VCs): formulas whose validity implies the code meets its contract. The VCs are discharged automatically by a SAT/SMT solver such as Z3.

Why it changed the field. Two decades of SMT progress automated *most* of the proof burden, lowering the barrier from "logician who can drive a proof assistant" to "engineer who can write good invariants." This is what put formal verification within reach of ordinary software teams.

The honest caveat. When a VC falls outside the solver's decidable theories (non-linear integer arithmetic is the classic offender, arising the moment you multiply two variables) the solver's heuristics may or may not close it. When they don't, the developer must supply a hint: an `assert`, an intermediate lemma, a manual step encoding the fact the solver missed. Automation is real but partial, and the human is pulled back in exactly at the hard spots. (This is also precisely where AI now helps; see Part V.)

Representative tools. Dafny (https://dafny.org), a verification-aware language compiling VCs to Z3, with **Verus** (https://verus-lang.github.io/verus/guide/), **Frama-C** (https://frama-c.com), and **Why3** (https://www.why3.org) in the same family.

4. Abstract interpretation and static analysis

Idea. Compute a *sound over-approximation* of all possible behaviors by executing the program over an abstract domain (intervals, signs, polyhedra) rather than concrete values. Because it over-approximates, it can prove the *absence* of whole error classes (no overflow, no null dereference) without enumerating runs, at the cost of false positives, flagging behaviors that cannot actually occur.

Trade-off. Highly automated and scalable, less expressive than the methods above. It excels at proving safety properties uniformly across a large codebase, which is why it is the workhorse of industrial static analysis.

5. The smart-contract-native toolchain

The methods above are general. The tokenization stack runs on the EVM, and the field has built a domain-specific toolchain that any practitioner will expect named, and that operates, crucially, at the level of **deployed bytecode**, not just source, closing the source-to-bytecode gap flagged in Part I:

— **Certora Prover** (https://www.certora.com), a commercial tool that has become one of the most widely used in production smart-contract verification, using its CVL specification language (https://docs.certora.com/en/latest/docs/cvl/overview.html) to express and prove contract invariants and rules against compiled code.
— **Halmos** (https://github.com/a16z/halmos) and **hevm** (https://github.com/ethereum/hevm), symbolic execution engines operating over EVM bytecode, proving or refuting assertions across all inputs.
— **K Framework** (https://kframework.org), rewrite-based executable semantics; the basis of **KEVM** (https://github.com/runtimeverification/evm-semantics), a complete formal semantics of the EVM itself, against which contracts can be verified.
— **Verity** (https://github.com/lfglabs-dev/verity), a Lean 4 framework for specified-and-verified contracts with EVM-oriented compilation.
— solc's built-in **SMTChecker** (https://docs.soliditylang.org/en/latest/smtchecker.html) and **Scribble** (https://github.com/Consensys/scribble) (runtime-assertion instrumentation) for lighter-weight, in-pipeline checking.
— **Slither** (https://github.com/crytic/slither) and similar for the static-analysis tier : fast, scalable detection of known bug patterns as a safety net.

The canonical properties this toolchain targets are domain-specific and worth naming, because they are what fails: **reentrancy safety** (no external call can re-enter and violate a balance invariant), **invariant preservation under arbitrary external calls and arbitrary call ordering**, **access-control correctness** (only authorized roles reach privileged state transitions), **arithmetic safety** (no overflow, correct rounding : the Bunni DEX was drained through a rounding error in audited code), **no unauthorized mint**, and **conservation of value** (tokens are neither created nor destroyed except by specified paths). These are real entries in the BPO catalogue introduced in Part IV (reentrancy is `BPO:0020`, access-control correctness `BPO:0040`, no-unauthorized-mint `BPO:0001`, conservation-of-value `BPO:0101`).

A map of the methods, and its load-bearing caveat

As a first approximation: model checking is *automatic but bounded*; theorem proving is *expressive but human-led*; refinement methods are *correct-by-construction but modeling-heavy*; auto-active

verification is *the productive middle* SMT made viable; abstract interpretation is *scalable but coarse*; and the EVM-native tools bring all of this down to bytecode. Real assurance arguments compose them : model-check the protocol design, deductively prove the trickiest core algorithm, auto-actively verify the implementation, prove contract invariants against bytecode with a symbolic/SMT-backed prover, and run static analysis across everything else.

Maturity: industrial deployment vs. scientific standing

These methods are not equally proven, and conflating "exists in the literature" with "battle-tested in production" is its own form of overselling. A rough, honest reading of where each sits on two separate axes : **scientific maturity** (theoretical foundations, decades of peer-reviewed development) and **industrial maturity** (routine use on real production systems by non-researchers):

Method	Scientific maturity	Industrial maturity	Notes on the gap
Static analysis / abstract interpretation	High	High	The most deployed by far; Astrée-class analyzers verify avionics, and lightweight analyzers are ubiquitous. The trade is precision, not readiness.
Model checking	High	Medium–High	Decades of theory; heavy use in *hardware* (Intel, etc.) and protocols. Lighter in mainstream software, where modeling cost bites.
Theorem proving (Coq/Rocq, Isabelle, Lean)	Very high	Low–Medium	The deepest foundations and the landmark results (seL4 kernel, CompCert compiler), but still expert-only and rarely in routine commercial pipelines.
Refinement / B-Method, Event-B	High	Medium (niche)	Genuine safety-critical deployments (rail signaling, aerospace), but concentrated in regulated industries with specialist teams; little mainstream-software footprint.
Auto-active / SMT-backed (Dafny, Verus, Why3)	High	Low–Medium, rising	Mature theory and improving usability; real but still limited industrial adoption, mostly at sophisticated shops. The trajectory is the story here.
Smart-contract FV (Certora, Halmos, hevm, KEVM)	Medium (young, fast-moving)	Medium, concentrated	The newest cluster. Real production use on high-value protocols, but adoption is uneven and the tools are evolving rapidly : promising rather than settled.

Two cautions follow from the table. First, the smart-contract toolchain, exactly the one the tokenization thesis leans on, is the *least mature* on both axes, which is an argument for the shared-specification and coverage discipline of Part IV, not against the tools. Second, the methods with the strongest guarantees (theorem proving, refinement) are the least industrially routine; the methods that are industrially routine (static analysis) give the weakest guarantees. There is no single method that is simultaneously high-assurance, automated, and widely deployed, which is why serious assurance composes several, and why "we formally verified it" should always prompt the question "*with which method, at what maturity, and covering which properties?*"

The "compose them" in the map above is also doing enormous work, and it is the field's hardest unsolved problem. We turn to it next, because pretending composition is free is how component-verified systems still get exploited.

Part III : The two things the toolbox is missing

Every method in Part II presupposes two things it does not itself provide: a *specification* to check against, and a way to *compose* local results into a system-level guarantee. Both gaps are where real systems fail.

Gap one: the specification problem

Every method takes "the property" as an input. None tells you *which* properties a system of this kind should have, what each means, how it tends to fail, or how it interacts with the others. In practice that knowledge is not amateurish or absent : mature audit firms hold deep internal checklists and methodologies. The problem is that this knowledge is **private, fragmented, and non-interoperable**: it lives in proprietary playbooks, post-mortems, and a few specialists' heads, and it does not port between firms, tools, or engagements. The consequences are structural:

- **Coverage depends on who showed up.** Each engagement reconstructs its requirement set from the reviewers' experience; two competent audits of the same contract can check materially different property sets and neither can prove it was complete.
- **Tools cannot be composed or compared.** A fuzzer's notion of a bug, a static analyzer's rule set, and a prover's specification share no common vocabulary, so their results cannot be unioned into a coverage claim.
- **Assumptions go unstated.** "We verified solvency" silently omits "...assuming the oracle is honest and the chain is live." Conditional assurance gets consumed as unconditional.
- **Conflicts surface only after the exploit.** An emergency-pause capability and a guaranteed-withdrawal property are in direct tension; transparency and privacy pull against each other. These trade-offs are known but nobody is *forced* to confront them at design time.

The fix is not another checking tool. It is a **shared, machine-readable map of what to check,** addressed in Part IV.

Gap two: the composition problem

Most of the methods in Part II verify *components*. But the large logic-driven exploits of recent years—oracle manipulation, cross-chain bridge failures, economic and MEV attacks—are rarely a single broken component. They are **emergent**: each contract is correct in isolation, and the failure lives in the *interaction*, often coupled to off-chain incentives the on-chain proof never modeled. A protocol can be a collection of individually verified contracts and still be exploitable as a system.

This is not a gap a better tool closes by itself, and it would be dishonest to imply otherwise. Composing local proofs into a global guarantee requires explicit **assume-guarantee reasoning** : each component's proof states what it assumes of its environment and what it guarantees in return, and the composition is sound only if every assumption is discharged by some other component's guarantee. Doing this at the scale of a live DeFi stack (contracts + bridges + oracles + governance + autonomous agents + economic actors) is, today, partially manual and substantially an open research problem. Two implications follow honestly:

- **Component verification is necessary but not sufficient.** Claims of safety must state their compositional scope: "this contract preserves its invariants under arbitrary calls" is not "this protocol is safe."

— **System-level and economic properties are partly outside current FV reach.** Incentive-compatibility, MEV-resistance, and liveness under adversarial validators need game-theoretic and economic analysis alongside formal proof, not instead of it.

A shared specification layer (Part IV) does not *solve* composition, but it is a precondition for attacking it: assume-guarantee reasoning is only tractable when assumptions and guarantees are written in a common vocabulary with stable identities. You cannot discharge an assumption against a guarantee if the two were never expressed in the same terms.

Part IV : The Blockchain Property Ontology (BPO)

The **Blockchain Property Ontology** (TKNFRA/blockchain-property-ontology) is a living, machine-readable catalogue of the properties blockchain systems are *supposed* to guarantee, and, for each, a precise account of what it means, why it matters, how it can fail, and how it can actually be verified. The animating analogy is a periodic table for correctness: every property gets a stable identity and a fixed position relative to every other, so the field can reason about correctness as a structured whole rather than a pile of anecdotes.

BPO is an open, machine-readable repository (dual-licensed: ontology content under CC BY-SA 4.0, schema and validator under Apache-2.0), authored and maintained by Zakaryae Boudi. What is public today is a deliberately compact *seed* : the schema, the classification scheme and relationship vocabulary, an automated validator, and a set of fully-worked exemplar properties chosen to stress every part of the design. The article describes that real artifact, not a hypothetical one.

A real worked entry: reentrancy safety (`BPO:0020`)

Rather than invent an illustrative property, here is the actual catalogue entry for reentrancy rendered from the repository (abridged; the live record carries more detail):

```
id: BPO:0020
slug: reentrancy-safety
title: Reentrancy Safety (State Consistency Across External-Call Boundaries)
status: draft
version: 0.1.0

classification:
  modal_class:        [safety, hypersafety]   # note: TWO modal readings
  deontic_role:       [invariant, guarantee]
  domains:            [evm, contract, access-control]
  abstraction_levels: [contract, bytecode]     # spec targets bytecode, not just source
  verification_modalities:
    [smt, symbolic-execution, model-checking, static-dataflow, runtime-verification]
  decidability: decidable-bounded

descriptions:        # one property, multiple readers, one referent
  semantic: >
    When a contract makes an external call (which may hand control to untrusted
    code), no re-entrant invocation can observe or exploit a transiently
    inconsistent internal state to extract value or violate an invariant.
  system_theoretic: >
    Two complementary readings — (1) a LOCK reading: a mutual-exclusion safety
    invariant; (2) an ATOMICITY reading: a 2-safety hyperproperty comparing the
    re-entrant schedule against the serialized one.
  formal_verification: >
    The lock formulation is a 1-safety state invariant, directly SMT-/symbolic-
    checkable. The atomicity formulation is a 2-safety hyperproperty requiring
    self-composition — which single-trace tools cannot establish.

formalization:
  first_order_logic: |
    Lock safety:  ∀ reachable s . ¬( inGuarded(s) ∧ aboutToEnterGuarded(s) ).
    Effects-before-interactions:  ∀ external-call site call_k .
      CriticalInv holds in the pre-state of call_k.
  temporal_logic:
    ltl: "G( externalCall → CriticalInv )  and  G( enterGuarded → ¬locked )"
    fairness: "None — pure safety."

attack_surface:
  - class: single-function-reentrancy
    incident_refs: ["The DAO (2016) — VERIFIED historical incident, canonical"]
```

```
  - class: cross-function-reentrancy
    notes: "A per-function lock is insufficient; the guard must cover the
            shared-state group."
  - class: read-only-reentrancy
    notes: "A view function returns stale state mid-callback, mispricing a
            dependent protocol."
  - class: cross-contract-reentrancy
    notes: "Requires reasoning about the whole trust cluster, not one contract."

verification:
  strategies:
    - approach: Invariant-at-call-site rule
      tools:    [Certora Prover, Halmos]
      strength: proof
    - approach: Reentrancy dataflow detection
      tools:    [Slither, Mythril]
      strength: heuristic-evidence       # high recall, some false positives
    - approach: Self-composition for atomicity
      tools:    [relational verifier, hevm]
      strength: bounded-proof

relationships:
  - {type: composesWith, target: "BPO:0101", note: "reentrancy is a principal
       mechanism by which conservation-of-value breaks (double withdrawal)"}
  - {type: mitigates,    target: "ATK:read-only-reentrancy"}

assumptions:           # first-class, and discharged by another property
  - kind: environmental
    statement: "The mutex storage cannot be reset out-of-band by a
                delegatecall/storage-collision path."
    discharged_by: "BPO:0030"            # storage-layout isolation
```

Several things in this *real* entry are worth drawing out, because they are exactly what the prose claims and they are present in the artifact rather than asserted about it:

— **One entry, many readers.** The `descriptions` block speaks to the auditor (`semantic`), the systems theorist (`system_theoretic`), and the formal-methods researcher (`formal_verification`); the `formalization` block hands a tool machine-readable FOL and LTL; the `verification` block hands a developer a strategy. One source of truth, many doors in — and no forking of meaning.

— **Honest about what each tool can deliver.** The entry explicitly splits the *lock* reading (a 1-safety invariant an SMT or symbolic tool can prove) from the *atomicity* reading (a **2-safety hyperproperty** that single-trace tools *cannot* establish at all, needing self-composition). Each verification strategy is tagged `proof`, `bounded-proof`, or `heuristic-evidence` — so "verified" never collapses into one undifferentiated word.

— **Assumptions are first-class and discharged.** The reentrancy guarantee openly rests on an assumption (the mutex can't be reset via a storage-collision path) and points to the other* property that discharges it (`BPO:0030`, storage-layout isolation). This is the assume-guarantee discipline of Part III, encoded as data — the precondition for composition.

— **Relationships, including conflict and composition, are edges in a graph.** `composesWith BPO:0101` records that reentrancy is a primary way conservation-of-value breaks; the catalogue's relationship vocabulary gives each edge a precise meaning (`implies` as denotational containment, `conflictsWith` as disjoint behavior sets, `dependsOn` as the assumption-premise relation, and so on).

— **Bytecode, not just source.** `abstraction_levels: [contract, bytecode]` ties the property to what actually executes, matching the source-to-bytecode caveat from Part I.

A necessary precision, so the multi-encoding point is not itself oversold: the entry gives FOL, LTL, Hoare, and state-machine encodings of one property, but BPO does not *mechanically prove* these encodings equivalent, that would be a hard formal result in its own right. What it provides is a **shared referent with a curated, human-validated correspondence**: everyone is demonstrably *trying to verify the same property*, drift between teams is prevented, and where cross-encoding faithfulness matters it becomes an explicit, auditable claim rather than a silent assumption. The validator enforces internal consistency—schema-checking every record, failing on any dangling relationship, and printing the ledger of accepted (undischarged) assumptions—but it does not, and does not claim to, prove semantic equivalence across logics.

Where BPO sits in the assurance lifecycle

BPO is not a checking tool and does not compete with the engines in Part II. It is the **scoping, specification, and coverage-attestation layer** around them, and it earns its place at three concrete points in the workflow:

— **Pre-engagement scoping.** Before an audit or verification effort, the relevant BPO entries for the system class define the *target property set*, turning "we'll review the contracts" into "we will establish these N properties, by these methods, under these assumptions." Completeness becomes assessable instead of implicit.

— **Coverage attestation, the part a CISO pays for.** This is BPO's strongest practical contribution. Against the catalogue, a deliverable can state: of the N properties this system class should hold, here are the ones established, by which method, at what scope (source vs. bytecode), under which assumptions, and, critically, *here is what remains open*. No individual tool can produce that statement, and it is exactly what an institution allocating capital to tokenized assets must see. A coverage report keyed to a shared ontology is portable across firms and comparable across protocols in a way that today's prose audit reports are not.

— **Continuous gating.** Because entries are machine-readable, the property set can become a CI gate: an upgrade that touches verified bytecode flags the entries whose proofs it invalidates, directly addressing the upgrade-path TCB risk from Part I.

On the conflict graph

Treating property conflicts as first-class is BPO's most novel and most systems-theoretic claim, and it should not be oversold. A conflict relation across a large property set raises real questions: is it pairwise or higher-order, is it transitive, who curates it, and does it stay tractable or degenerate into an unmaintainable tangle as the catalogue grows? The defensible position today is that the conflict graph is *valuable even if incomplete* : surfacing the well-understood tensions the catalogue already encodes (the emergency-pause property `BPO:0050` against the eventual-withdrawal guarantee `BPO:0002`; transparency against privacy) already prevents recurring classes of design error, while its completeness and curation model are genuinely open and a subject for ongoing work. Naming that limit is itself in keeping with BPO's discipline: state the assumptions, including the assumptions about BPO.

The bridge to agent verification

BPO also grounds the verification work Tokenfrastructure has developed for AI agents in tokenized financial protocols, where an ontology-powered framework grounds agent decisions in a formal ontology, constrains agent behavior with formal models (invariants, preconditions, allowed state transitions), and produces cryptographically anchored, independently verifiable execution proofs. There the ontology is not documentation; it is the semantic substrate that makes a non-deterministic agent's behavior amenable to deterministic, checkable guarantees, turning the runtime, behavioral verification problem raised in the intro into something with a fixed vocabulary to verify against. BPO is the general-purpose form of that same move: ontology as the precondition for verification and for composition at scale.

Part V : Where AI genuinely helps scale verification, and where it is itself a risk

Formal verification's adoption ceiling has always been two walls: **the expertise required** and **the manual effort per artifact**. AI is now demonstrably good at lowering specific bricks in both, but only inside an architecture that never has to trust it, and that architecture has a sharp internal boundary that must be stated precisely, because the field's most seductive error is to apply the comfort of one layer to another where it does not hold.

1. Proof and hint synthesis : strong, measured, and soundly contained

The clearest win is automating the manual hints auto-active tools demand. **DafnyBench**—over 750 programs, ~53,000 lines—tests whether LLMs can generate enough hints for Dafny's engine to verify them; the best model-and-prompt combination reached a 68% success rate *as of the original benchmark* (a GPT-4 / Claude-3-era result), improving with error-message retry and degrading as required hints grow. The headline number matters less than the trajectory: it has moved since, and the point of this section is the *trend*, not the snapshot.

Two lessons. First, AI is already useful exactly at the SMT-can't-close-this-VC spot where humans were previously indispensable. Second, the model works *inside a loop with a sound checker*. The LLM proposes a hint; the verifier disposes. At the proof layer, the AI never has to be trusted, because a wrong hint simply fails to verify. Systems operationalize this: **Clover** generates Dafny code and annotations, **Lemur** wraps LLM reasoning around verification frameworks, **Laurel** generates and checks specifications, and LLM-guided proof search is pushing into **TLA+** and **Lean**. The pattern is *generator plus verifier*, with the verifier's soundness containing the model's unreliability.

2. Specification drafting : high-leverage, and where the soundness comfort does NOT transfer

Here is the boundary the field must not blur. The proof-layer comfort ("a wrong AI output just fails to verify") **is false at the specification layer.** A wrong spec verifies *fine*; the prover dutifully proves the wrong thing, and the green checkmark is now actively misleading. AI is genuinely well-suited to *drafting* specs (reading code and intent to propose candidate `requires/ensures/invariant` clauses or temporal properties) and this attacks the real bottleneck rather than the prover. But the discipline that contains it cannot be the verifier. It must be (a) a canonical ontology (BPO) against which a drafted spec is checked for conformance to what the property is *supposed* to mean, and (b) human ratification. The precise statement: soundness contains AI unreliability at the proof layer; at the spec layer, only the ontology and a human do. Conflating the two is how AI-assisted verification could manufacture false assurance at scale.

3. Counterexample triage and abstraction discovery

When a model checker returns a trace or a static analyzer floods you with candidates, AI is well-suited to summarizing traces, separating likely-real findings from false positives, and proposing the abstractions that tame state-space explosion (recall: those abstractions are the human-supplied step that makes model checking "automatic"). None of this needs to be sound to be useful ; it is a productivity layer in front of sound tools, and a wrong suggestion costs time, not assurance.

4. Ontology engineering itself

A recursive opportunity: AI can accelerate building and maintaining the ontology that, in turn, disciplines AI's contributions to verification. The literature on LLM-assisted ontology engineering is candid, real acceleration of drafting and extraction, against robust limits on consistency, coverage, and validation. The framing that holds: AI as drafter, ontology as validation harness, human as governor. Ontologies are not made obsolete by LLMs; they become the scaffolding that keeps generative pipelines honest.

5. The new attack surface: AI in the loop is itself a target

The intro raised agent manipulability as a *motivating* threat; intellectual honesty requires closing that loop, because putting AI inside the verification pipeline creates its *own* adversarial surface that the "sound verifier contains everything" story does not fully cover:

- **Spec-layer poisoning.** Since a wrong spec verifies cleanly, an adversary who can influence AI-drafted specifications, through a poisoned training signal, a compromised generator, or adversarial in-context input, can steer a system toward proving a subtly weakened property. The proof layer will not catch it; only ontology conformance and human review will. This is the spec-layer boundary from §2, weaponized.
- **Ontology poisoning.** If AI helps maintain BPO and a malicious or erroneous entry enters the catalogue (a quietly weakened invariant, a dropped assumption, a missing conflict edge) it corrupts the reference everything else is checked against. The ontology's integrity therefore needs the same first-class treatment it demands of everything else: provenance, review, and version-controlled, signed changes.
- **Agent input and control-path manipulation.** For runtime agents (the B-Method-constrained agents of Part IV), the verified property only holds while the agent's inputs and control paths are within the modeled assumptions. An attacker who manipulates those inputs operates in the *unverified* gap between the model's assumptions and reality : the same "assumptions may not match reality" risk from Part I, now adversarial and live.

The defense is not to keep AI out; it is to treat every AI-touched artifact as untrusted input to a sound, ontology-anchored, human-governed pipeline, and to extend provenance and integrity guarantees to the ontology and the spec layer, not just the proof.

The architecture that scales and its hard limits

All five threads compose into one principle:

> **AI proposes (hints, specs, abstractions, ontology drafts) and a sound, ontology-anchored, human-governed pipeline disposes. Soundness contains AI at the proof layer; the ontology and human ratification contain it at the spec layer; provenance and review contain it at the ontology layer.**

This is powerful precisely because it never asks anyone to trust the AI. But the limits must be stated as plainly as the promise:

- **AI cannot certify its own output.** Remove the sound checker and trust the model's "this is verified," and you have replaced a proof with a vibe.

— **AI cannot tell you whether the spec is the right spec**. It proposes; only a human consulting a shared ontology ratifies. A perfect proof of a wrong spec is a liability dressed as an asset.

— **AI does not solve composition.** Faster component proofs do not assemble themselves into a system-level guarantee; assume-guarantee reasoning across a live stack remains substantially open (Part III).

— **Liveness and the economic layer resist the easy wins.** A safety property closed by AI-suggested invariants is one thing; liveness in a permissionless system depends on fairness assumptions (censorship-resistance, honest-validator fractions, chain liveness under attack) that are *exactly* what adversaries target, and that need proof plus economic analysis, not fuzzing dressed up by a confident summary.

— **Coverage is not correctness.** AI can raise verification *throughput* dramatically while leaving *what should be verified* as under-specified as before. This is the gap BPO exists to close.

The synthesis is not "AI replaces formal methods" and not "AI is irrelevant to them." It is that a shared property ontology and AI are **two complementary multipliers** the discipline has lacked, sitting on either side of the checking machinery it already has. The ontology fixes *what* we verify, *what the words mean*, and *what was assumed*; AI accelerates the *labor* of getting it verified, inside a loop where soundness is never delegated and the spec layer is never left unguarded. Neither alone suffices. Together — ontology steering, AI generating, sound verifiers gating, humans governing — they are how formal verification can move from a boutique practice for a handful of experts to standard infrastructure for systems that, very soon, will be too valuable to fail.
