Six-Layer Formal Verification of a Safety-Critical AI Governance Kernel

John McGraw
TaskHawk Systems, LLC
Charlottesville, Virginia
j.mcgraw@taskhawktech.com

March 2026

Abstract

Agentic AI systems that initiate financial transactions, invoke external services, and modify production infrastructure require enforcement boundaries that are independently verifiable. We present the first six-layer formal verification of a single AI governance product: the Kevros Enforcement Kernel, a permission-before-power architecture that mediates between autonomous agents and consequential action. Using TLA+/TLC (1.94 billion states, 12 safety invariants, 4 liveness properties), Amazon Kani/CBMC (17 of 17 bounded model checking harnesses), Microsoft Verus/Z3 (22 of 22 unbounded proofs), runtime assertion checking, cross-language golden vectors, and Lean 4 interactive theorem proving (20 theorems, 0 `sorry`), we verify 12 safety properties and 4 liveness properties across all abstraction levels for under $4 in compute time. The combined result is 71 proofs with zero failures across six independent verification tools. We document four engineering contributions: (1) an abstraction technique that eliminates IEEE 754 bit-blasting from bounded model checking by replacing floating-point thresholds with algebraic region discriminants, reducing SAT solving time from over 35 minutes to 33 seconds; (2) a bisimulation relation that enables the same abstract model to be verified by both bounded (SAT) and unbounded (SMT) solvers; (3) evidence that modeling cryptographic hash functions as uninterpreted functions in Z3 yields proofs that hold for any hash algorithm; and (4) a machine-checked Lean 4 formalization that independently re-proves all 12 safety invariants and 4 liveness properties by induction on the reachability relation, with zero `sorry` and zero external dependencies. The TLA+ specification, Lean 4 formalization, and verification manifest are publicly available.

1 Introduction

AI agents are crossing the threshold from advisory to autonomous. Systems that execute financial transactions, modify procurement records, invoke external APIs, and control physical actuators cannot be governed by prompt engineering or policy documents alone. The enforcement boundary between the agent and consequential action is a safety-critical component. It requires the same verification rigor applied to avionics envelope protection, automotive safety monitors (ISO 26262), and reactor protection systems.

Current AI governance approaches focus on model alignment, output filtering, and organizational policy. These are necessary but do not provide verifiable guarantees at the actuation boundary. The NIST AI Risk Management Framework [5] provides governance structure but does not specify technical enforcement mechanisms. The OWASP Top 10 for Agentic Applications [6] identifies tool misuse and delegation without attenuation as top-tier risks. No published standard requires formal verification of enforcement logic.

The central insight of this work is architectural: *you do not need to verify the model to verify the safety of the system.* The enforcement kernel is a finite, deterministic, inspectable component positioned between the model and consequential action. The model is treated as an untrusted input source. The kernel is the verified component. This separation is the same pattern used in safety-critical engineering, where the safety monitor is verified independently of the system it monitors.

We present the first six-layer formal verification of a single AI governance product:

1. **TLA+/TLC**: System-level model checking (1.94 billion states, 12 safety invariants, 4 liveness properties)
2. **Kani/CBMC**: Bounded model checking of the Rust implementation (17 of 17 harnesses verified)
3. **Verus/Z3**: Unbounded deductive proof of the Rust implementation (22 of 22 proofs verified)
4. **Runtime Assertion Checking**: 1:1 mapping of TLA+ properties to production Python code
5. **Golden Vectors**: Cross-language byte-identity equivalence between Python and Rust implementations
6. **Lean 4**: Interactive theorem proving — machine-checked mathematical proof of all safety and liveness properties (20 of 20 theorems, 0 `sorry`)

The combined result is 71 proofs with zero failures across six independent verification tools, at a total compute cost under $4. The TLA+ specification (716 lines), Lean 4 formalization (760 lines), and a machine-readable verification manifest are publicly available.[1]

2 Architecture: Permission-Before-Power

2.1 System Overview

The Kevros Enforcement Kernel implements a sequential pipeline of five components, each of which must succeed for non-zero motor output to occur:

1. **Mode Manager**: A four-state safety machine (`INERT`, `BOOT`, `RUN`, `FAULT`) with hysteresis thresholds governing transitions. Only the `RUN` state permits actuation-enabling decisions.
2. **Enforcer**: A sequential decision pipeline producing one of four verdicts: `ALLOW`, `CLAMP`, `DENY`, or `HALT`.
3. **Evidence Chain**: An append-only, hash-chained provenance ledger. Every decision is recorded before authorization can proceed.
4. **Release Token**: An HMAC-SHA256 signed authorization artifact issued only for `ALLOW` or `CLAMP` decisions with successfully written evidence.
5. **Actuation Gate**: A permission-before-power interlock that independently reconstructs the token preimage and performs constant-time HMAC verification before permitting motor output.

The pipeline processes requests through seven sequential phases:

$$\texttt{IDLE} \rightarrow \texttt{AUTH} \rightarrow \texttt{MODE\_CHECK} \rightarrow \texttt{DECIDE} \rightarrow \texttt{EVIDENCE} \rightarrow \texttt{TOKEN} \rightarrow \texttt{RESPOND}$$

[1] https://github.com/taskhawk-systems/kevros-formal-verification

2.2 Fail-Closed Design

The default state is `DENY` with zero actuation. Every failure at any stage (authentication, mode check, evidence write, token issuance, token verification) results in `DENY`. No code path exists from any failure condition to non-zero motor output. This architectural choice is foundational to verification tractability: the system is safe by default, and verification must show only that no transition sequence can violate this property.

2.3 Verification Boundary

The model (LLM, ML risk scorer) is not inside the verification boundary. It is an untrusted input. The safety confidence value (`safety_conf`) provided by the model is treated as a nondeterministic input drawn from its full range in every TLA+ state. This means the verified properties hold regardless of model behavior, including adversarial, miscalibrated, or compromised models.

3 Safety Properties

We define 12 safety invariants (state predicates verified across all reachable states) and 4 liveness properties (temporal properties verified under fair scheduling). Table 1 lists the safety invariants.

Table 1: Safety invariants verified across all six layers.

ID	Property	Description
SP1	PermissionBeforePower	No actuation without valid release token
SP2	FailClosedAuth	Any authentication error produces `DENY`
SP3	FailClosedEvidence	Any evidence write error produces `DENY`
SP4	EvidenceBeforeToken	Token issued only after evidence is appended
SP5	TokenRequiredForActuation	Actuation gate rejects without valid token
SP6	ModeDecisionConsistency	`ALLOW`/`CLAMP` only in `RUN` mode
SP7	FaultIsLatched	`FAULT` exits only to `INERT` via manual reset
SP8	NoDirectInertToRun	Must pass through `BOOT` before `RUN`
SP9	BootPromotionValid	`BOOT` to `RUN` requires N consecutive healthy ticks
SP10	BootTimeoutBounded	`BOOT` phase has bounded timeout
SP11	DenyZeroActuation	`DENY` or `HALT` produces zero actuation
SP12	ChainIntegrityInvariant	Evidence chain hash links are unbroken

The four liveness properties ensure progress under fair scheduling: the system eventually reaches `RUN` if conditions persist (LP1), eventually leaves `RUN` if safety drops (LP2), `BOOT` eventually completes (LP3), and the pipeline eventually returns to `IDLE` (LP4).

Properties SP1 and SP5 together establish the foundational guarantee: non-zero motor output requires both a validly issued release token *and* successful independent verification of that token by the actuation gate. SP4 ensures every authorization has a provenance record. SP2, SP3, and SP11 ensure that all failure paths terminate in `DENY` with zero actuation.

4 Verification Layers

4.1 Layer 1: TLA+/TLC (System Architecture)

The TLA+ specification models the complete enforcement kernel in 716 lines. It defines 20 state variables across five component groups, 15 actions (9 mode transitions, 7 pipeline phases, 1 environment step), and 12 safety invariants checked as state predicates across all reachable states.

Table 2: TLA+/TLC model checking results.

Metric	**Value**
States generated	1,943,069,194
Distinct states	171,647,834
Search depth	118
Safety invariants	12/12 verified
Liveness properties	4/4 verified
Violations	0
Runtime	18 minutes 52 seconds

The state constraint bounds `tick` $<$ 25, `epoch` $\leq$ `MAX_EPOCH`, and `chain_length` $\leq$ `MAX_EPOCH` + 1, producing a finite but large state space. The nondeterministic treatment of `safety_conf` (drawn from 0..10 at each tick), `time_oracle_ok`, and `integrity_ok` (each drawn from `BOOLEAN`) ensures that every possible environment condition is explored.

For context, AWS has published TLA+ results for DynamoDB, S3, and EBS, with state counts ranging from thousands to tens of millions [1]. The Kevros specification at 1.94 billion generated states is among the largest publicly documented TLA+ model checking runs. The SysMoBench benchmark (2025) demonstrated that current AI systems, including GPT-4 and Claude, cannot reliably generate correct TLA+ specifications for complex real-world systems, underscoring that specifications at this level of complexity require manual authoring [7].

4.2 Layer 2: Kani/CBMC (Bounded Model Checking)

Kani is Amazon's open-source bounded model checker for Rust [2]. It translates Rust MIR into CBMC format and uses SAT/SMT solving to exhaustively verify properties up to a specified bound. Amazon uses Kani on Firecracker (27 harnesses across 3 verification suites), s2n-quic, and s2n-tls.

We wrote 17 Kani harnesses mapping to all 12 safety properties. Cryptographic functions (`ring::digest`, `ring::hmac`) are stubbed with `#[cfg(kani)]` conditional compilation, modeling crypto as nondeterministic functions with correct type signatures. This is necessary because CBMC cannot efficiently reason about FFI calls into C and assembly implementations.

Table 3: Kani/CBMC verification results.

Metric	Value
Harnesses total	17
Harnesses verified	17
Harnesses failed	0
Unit tests passed	99
Build phase	33 seconds
Environment	AWS CodeBuild 2XLARGE (72 vCPU, 145 GB)

Three sequential engineering challenges arose during verification: shell syntax incompatibilities in the CodeBuild environment, dynamic memory allocation patterns incompatible with CBMC, and IEEE 754 floating-point bit-blasting (Section 5.1). The solutions to these challenges constitute primary engineering contributions of this paper.

4.3 Layer 3: Verus/Z3 (Unbounded Proof)

Verus is Microsoft Research's deductive verification tool for Rust [3]. Where Kani checks properties up to a finite bound, Verus proves properties hold for *all* possible inputs and *all* possible execution lengths.

We wrote 22 proofs across three source files:

Table 4: Verus/Z3 proof results.

File	Proofs	Properties	Technique
`mode_manager.rs`	7	SP7, SP8, SP9, SP10	Exhaustive match
`pipeline.rs`	9	SP1–SP6, SP11	All input combinations
`evidence.rs`	6	SP12 (3 sub-properties)	Uninterpreted hash
Total	**22**	**All 12 SPs**	

Total Z3 verification time was 2 seconds. The proofs use the same `AbstractModeManager` and `KappaRegion` abstraction created for Kani (Section 5.2), establishing a bisimulation relation between the bounded and unbounded verification layers.

The evidence chain proofs (SP12) model the hash function as an uninterpreted function in Z3 (Section 5.3). This yields proofs that hold for *any* hash algorithm satisfying the functional property that equal inputs produce equal outputs.

4.4 Layer 4: Runtime Assertion Checking

The Python production implementation maps each TLA+ safety property to a runtime assertion checked on every enforcement cycle. Violations trigger `DENY` with an alert, maintaining the fail-closed invariant. This layer provides continuous verification against live production traffic, complementing the static analyses of Layers 1–3.

4.5 Layer 5: Golden Vectors

The Rust and Python implementations must produce byte-identical outputs for all decision paths. A suite of 23 test vectors covers all enforcement verdicts, mode transitions, evidence chain operations, and edge cases. Verification time is under 1 second. This layer ensures that the Rust implementation (verified by Kani and Verus) is functionally equivalent to the Python implementation deployed in production.

4.6 Layer 6: Lean 4 (Interactive Theorem Proving)

Lean 4 is an interactive theorem prover developed at Microsoft Research [8]. Unlike TLC (bounded exhaustive search), Kani (bounded model checking), or Verus (automated SMT), Lean 4 requires the author to construct explicit proof terms that are independently verified by a small trusted kernel. The resulting proofs are machine-checked mathematical certificates: every logical step is validated, and the proof cannot contain gaps.

We wrote a 760-line Lean 4 formalization (`KevrosCorrect.lean`) that faithfully translates the 716-line TLA+ specification into Lean's type theory. The formalization independently re-proves all 12 safety invariants (SP1–SP12) and establishes structural liveness for all 4 liveness properties (LP1–LP4).

Table 5: Lean 4 theorem proving results.

Metric	**Value**
Theorems total	20
Theorems verified	20
`sorry` count	0
Safety theorems	14 (inv_holds, safety_holds, SP1–SP12)
Liveness theorems	6 (pipeline_progress, LP1–LP4)
Source lines	760
External dependencies	0 (pure Lean 4.15.0+)

4.6.1 Proof Architecture.

The formalization defines the system state as a Lean `structure` with 18 fields mirroring the TLA+ state variables. The initial state predicate (`IsInit`) and the next-state relation (`Next`) are defined as an inductive type with 24 constructors, one for each TLA+ action. Every constructor fully constrains all 18 fields of the successor state, ensuring frame conditions are explicit.

Reachability is defined inductively:

```
inductive Reachable (nP bT dT : Nat) : State -> Prop
  | init (s : State) (h : IsInit s) : Reachable nP bT dT s
  | step (s s' : State) (hR : Reachable nP bT dT s)
      (hN : Next nP bT dT s s') : Reachable nP bT dT s'
```

The safety invariant (`SafetyInv`) is a conjunction of 12 predicates corresponding exactly to SP1–SP12. A strengthened invariant (`Inv`) adds 15 auxiliary conjuncts required for inductive closure, yielding 27 total conjuncts.

4.6.2 Main Theorem.

The central result is `inv_holds`: for all parameters `nP`, `bT`, `dT` and any state `s` reachable from an initial state, the strengthened invariant `Inv s nP bT dT` holds. The proof proceeds by induction on the `Reachable` relation:

- **Base case**: `IsInit` satisfies all 27 conjuncts (all fields fully constrained by the initial state predicate).
- **Inductive case**: For each of the 24 `Next` constructors, rewrite the successor state fields using the constructor hypotheses and appeal to the induction hypothesis. This produces approximately $24 \times 27 = 648$ proof obligations, each resolved mechanically by `simp_all` with targeted `omega` reasoning for natural-number subtraction (SP10) and case analysis for decision types (SP11 in `rspApply`).

The safety invariant follows immediately: `safety_holds` extracts the first component of the strengthened invariant. Each SP1–SP12 is then stated and proved as an individual theorem by projection.

4.6.3 Liveness.

TLA+ liveness uses temporal logic (leads-to $\rightsquigarrow$) under weak fairness. In Lean, we encode liveness as *bounded progress*: we define a well-founded variant and prove it strictly decreases on relevant transitions.

For LP4 (Pipeline Completes), we define a variant function `phaseDist` from `Phase` to `Nat`, assigning distances: `idle` $\mapsto 0$, `respond` $\mapsto 1$, ..., `auth` $\mapsto 6$. Theorem `pipeline_progress` shows that every non-idle phase has a `Next` successor with strictly smaller `phaseDist`. Theorem `LP4_PipelineCompletes` then uses well-founded induction on `phaseDist` to conclude that the pipeline reaches `IDLE` in at most 6 steps.

LP1 (Eventually RUN) is decomposed into two structural lemmas: `LP1_InertToBoot` shows that `INERT` transitions to `BOOT` under good conditions, and `LP1_BootProgress` shows that each `BOOT` step either promotes to `RUN` or increments `consecutiveOk` toward promotion. LP2 and LP3 are proved as single-step existential lemmas under their respective trigger conditions.

This encoding is weaker than TLA+'s universal quantification over fair traces but sufficient for establishing that progress is structurally possible — and it is machine-checked with zero `sorry`.

4.6.4 Key Design Decisions.

- **KappaRegion abstraction**: The same three-region enum (`belowOff`, `between`, `aboveOn`) used in Layers 2 and 3, eliminating IEEE 754 from the proof.
- **Uninterpreted hash**: Chain integrity is modeled as a boolean (`chainIntact : Bool`) rather than concrete hash computation. Proofs hold for any hash function.
- **Zero dependencies**: Pure Lean 4.15.0+ with no Mathlib imports and no axioms beyond the Lean kernel. Verification requires only `lake build`.
- **Faithful translation**: One-to-one correspondence with TLA+ actions (each `Next` constructor cites the corresponding TLA+ line number).

4.6.5 Relationship to Other Layers.

Layer 6 is complementary to, not redundant with, the other verification layers. TLA+ (Layer 1) exhaustively checks a bounded state space; Lean 4 proves properties hold for *unbounded* parameters. Kani (Layer 2) verifies the Rust implementation; Lean 4 verifies the mathematical model. Verus (Layer 3) uses automated SMT; Lean 4 uses human-guided interactive proof with a smaller trusted computing base. The agreement of all six layers on the same 12 safety properties provides defense in depth: a bug would have to simultaneously escape exhaustive model checking, bounded Rust verification, unbounded SMT proof, runtime assertion checking, cross-language byte equivalence, *and* machine-checked interactive theorem proving.

5 Engineering Contributions

5.1 Eliminating IEEE 754 from Bounded Model Checking

CBMC represents IEEE 754 double-precision values as 64-bit bitvectors: 1 sign bit, 11 exponent bits, and 52 mantissa bits. Every threshold comparison (e.g., `safety_conf >= THETA_ON`) generates millions of SAT clauses because the solver must reason about all possible bit patterns. The `mode_manager_composite_safety` harness ran for over 35 minutes on a 72-vCPU instance before timeout.

The solution replaces floating-point comparisons with an algebraic region discriminant. Instead of comparing `f64` values against `f64` thresholds, we classify the input into three regions:

```
enum KappaRegion {
    BelowOff,  // kappa < THETA_OFF
    Between,   // THETA_OFF <= kappa < THETA_ON
    AboveOn,   // kappa >= THETA_ON
}
```

The mode manager's transition logic depends only on which region the safety confidence falls into, not on the precise floating-point value. The `KappaRegion` enum requires approximately 2 SAT variables per comparison, compared to approximately 2,000 for IEEE 754 doubles. Build time dropped from over 35 minutes per harness to 33 seconds for all 17 harnesses combined.

This technique is applicable to any bounded model checking target that uses floating-point thresholds for discrete control decisions. The state machine transitions are identical. The invariants are identical. The only change is the representation of the input domain. We believe this is the first documented application of this specific abstraction to Kani/CBMC.

5.2 Bisimulation Across Verification Tools

The `AbstractModeManager` with `KappaRegion` serves both Kani (bounded, SAT) and Verus (unbounded, SMT). The simulation relation is defined as follows: the abstract model and the concrete `ModeManager` share the same state set (`INERT`, `BOOT`, `RUN`, `FAULT`), the same transition edges (governed by region membership rather than precise threshold values), and the same reachable states (verified by TLC at the TLA+ level).

Kani verifies properties for all inputs within its bounds. Verus verifies properties for all possible inputs without bounds. The proofs are complementary, not redundant: Kani can verify properties involving unsafe code and FFI boundaries that Verus cannot reach, while Verus provides mathematical guarantees that Kani cannot because bounds are always finite.

The ability to share a single abstraction across two independent verification backends with different solver technologies (SAT and SMT) reduces the total proof engineering effort. The abstraction was designed once for Kani and reused without modification for Verus.

5.3 Hash Functions as Uninterpreted Functions

The evidence chain (SP12) uses SHA-256 hash chaining in production. Modeling SHA-256 concretely in Z3 is intractable. Instead, we model the hash as an *uninterpreted function*: Z3 treats it as an oracle satisfying only the functional property (equal inputs produce equal outputs) and the type signature (`[u8; N]` $\rightarrow$ `[u8; 32]`).

The resulting proofs hold for *any* hash function satisfying this property: SHA-256, BLAKE3, SHA-3, or any post-quantum hash. If the hash algorithm is changed in production, the chain integrity proofs remain valid without re-verification. All quantifiers in the evidence proofs are universal (for all), requiring no existential witnesses.

The same approach is used in the Lean 4 formalization (Layer 6), where chain integrity is modeled as a boolean (`chainIntact : Bool`) that is preserved by all transitions.

5.4 Machine-Checked Inductive Invariant via Strengthening

The Lean 4 proof (Section 4.6) required strengthening the safety invariant from 12 conjuncts to 27 conjuncts to achieve inductive closure. The 15 auxiliary conjuncts capture pipeline-phase-specific facts (e.g., "at phase `AUTH`, decision is `DENY`" or "at phase `DECIDE`, mode is `RUN`") that are consequences of the transition structure but are needed by the induction hypothesis to close certain cases.

This strengthening process — identifying which auxiliary facts the inductive step requires — is the primary intellectual contribution of the Lean 4 formalization. Once the strengthened invariant is identified, the proof of approximately 648 obligations is largely mechanical. The `simp_all` tactic resolves most cases, with `omega` handling natural-number arithmetic for SP10 (boot timeout bounding) and case analysis handling decision-type reasoning for SP11 in the `rspApply` transition.

6 Comparison to Published Work

Table 6 compares the Kevros verification stack to published formal verification at AWS and Microsoft.

Table 6: Comparison of published formal verification across organizations.

Org.	Tools Published	Applied To	Unified
AWS	TLA+, CBMC, Property testing, Dafny	DynamoDB, S3, Firecracker, Encryption SDK	No
Microsoft	TLA+, Static Driver Verifier, Verus	Cosmos DB, Windows, research prototypes	No
Kevros	TLA+, Kani/CBMC, Verus/Z3, RAC, Golden Vectors, Lean 4	Kevros Enforcement Kernel (one product)	**Yes**

AWS and Microsoft have demonstrated individual verification tools on individual products at scale. Neither has published evidence of running six independent verification tools on a single product. The key distinction is not the tools themselves but the unified application: the same 12 safety

properties are verified at every abstraction level, from system architecture (TLA+) to bounded implementation (Kani) to mathematical proof (Verus) to machine-checked theorem (Lean 4) to production runtime (RAC) to cross-language equivalence (Golden Vectors).

AWS reported that formal methods added 13% overhead to the ShardStore project (S3) [1]. Our total verification cost was under $4 in compute. The low cost is attributable to four factors: (1) the enforcement kernel is deliberately small and finite-state, (2) the `KappaRegion` abstraction eliminated the dominant cost factor (IEEE 754 bit-blasting), (3) unbounded proofs via Verus/Z3 complete in seconds because Z3 reasons symbolically rather than enumerating states, and (4) the Lean 4 formalization requires only `lake build` with zero external dependencies.

7 Discussion

The results demonstrate that formal verification of AI enforcement boundaries is practical, affordable, and achievable with existing tools. Four implications follow.

First, standards bodies should distinguish between *governance controls* (policies, processes, documentation) and *enforcement controls* (technical mechanisms that prevent unsafe actions at runtime). Both are necessary. Neither is sufficient alone. Current guidance is heavily weighted toward governance. The tools and methodology described here provide a path toward requiring formal verification evidence for safety-critical agent subsystems.

Second, the architectural pattern of treating the model as an untrusted input makes formal verification tractable. Attempting to verify the model itself (a foundation model with billions of parameters) is currently infeasible. Verifying the enforcement boundary (a finite-state machine with 20 variables) is a solved problem. The safety guarantees hold regardless of model behavior because the model's output is an input to the verified kernel, not a trusted component.

Third, the cost barrier to formal verification is lower than commonly assumed. The Kevros verification stack was completed in under $4 of compute time. The primary engineering cost was not running the tools but designing the abstractions (particularly the `KappaRegion` elimination of floating-point arithmetic). Once designed, the abstractions are reusable across verification tools and will persist across implementation changes.

Fourth, interactive theorem proving (Lean 4) provides a qualitatively different assurance than automated methods. TLC, Kani, and Verus rely on complex automated backends (explicit-state search, SAT/SMT solving) whose implementations are large and potentially buggy. Lean 4's trusted computing base is a small type-checking kernel. A proof that type-checks in Lean is a mathematical certificate that can be independently audited. The addition of Layer 6 closes the assurance gap between automated verification (which is fast but relies on tool correctness) and deductive proof (which is slower to author but relies on a minimal trusted base).

8 Conclusion

We have presented the first six-layer formal verification of a single AI governance product. The Kevros Enforcement Kernel has been verified at six independent abstraction levels: system architecture, bounded implementation, unbounded proof, runtime assertion, cross-language equivalence, and machine-checked interactive theorem proving. Twelve safety properties and four liveness properties hold across 1.94 billion reachable states, 17 bounded model checking harnesses, 22 unbounded SMT proofs, and 20 Lean 4 theorems with zero `sorry` — totaling 71 proofs with zero violations.

The enforcement boundary, not the model, is where safety guarantees can and should be established. The tools exist. The methodology is established. The cost is negligible. We encourage standards bodies and procurement authorities to consider requiring formal verification evidence for safety-critical agent subsystems in high-consequence deployment contexts.

Availability

The TLA+ specification, Lean 4 formalization, TLC configuration, model checking output, and machine-readable verification manifest are available at:

https://github.com/taskhawk-systems/kevros-formal-verification

RFC 3161 timestamps from FreeTSA provide independent, cryptographic proof of the existence and content of all artifacts at the time of verification.

References

[1] C. Newcombe, T. Rath, F. Zhang, B. Munteanu, M. Brooker, and M. Deardeuff. How Amazon Web Services uses formal methods. *Communications of the ACM*, 58(4):66–73, 2015.

[2] A. Chong, N. Coughlin, B. Machado, and others. Code-level model checking in the software development workflow. *Proceedings of ICSE-SEIP*, 2021.

[3] A. Lattuada, T. Hance, C. Cho, M. Brun, I. Subasingha, Y. Zhou, J. Howell, B. Parno, and C. Hawblitzel. Verus: Verifying Rust programs using linear ghost types. *Proceedings of the ACM on Programming Languages (OOPSLA)*, 2023.

[4] L. Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002.

[5] National Institute of Standards and Technology. Artificial Intelligence Risk Management Framework (AI RMF 1.0). NIST AI 100-1, January 2023.

[6] OWASP Foundation. OWASP Top 10 for Agentic Applications. December 2025.

[7] SysMoBench Consortium. Can AI Generate Correct TLA+ Specifications? A Benchmark Study. 2025.

[8] L. de Moura and S. Ullrich. The Lean 4 theorem prover and programming language. *Proceedings of CADE-28*, Lecture Notes in Computer Science, vol. 12699, 2021.

A Safety Property Definitions (TLA+)

The following are the formal TLA+ definitions of the 12 safety invariants.

Pipeline Invariants (SP1–SP6):

```
(* SP1: Permission-Before-Power *)
PermissionBeforePower ==
    actuation = "APPLIED" => token_verified

(* SP2: Fail-Closed Authentication *)
FailClosedAuth ==
    (phase \in {"EVIDENCE", "TOKEN", "RESPOND"}
    /\ ~request_hmac_ok) => decision = "DENY"

(* SP3: Fail-Closed Evidence *)
FailClosedEvidence ==
    (phase \in {"TOKEN", "RESPOND"}
    /\ ~evidence_written /\ ~write_succeeded)
        => decision = "DENY"

(* SP4: Evidence Before Token *)
EvidenceBeforeToken ==
    token_issued => evidence_written

(* SP5: Token Required for Actuation *)
TokenRequiredForActuation ==
    actuation = "APPLIED" => token_issued

(* SP6: Mode-Decision Consistency *)
ModeDecisionConsistency ==
    (phase \in {"EVIDENCE", "TOKEN", "RESPOND"}
    /\ decision \in {"ALLOW", "CLAMP"}) => mode = "RUN"
```

Mode and System Invariants (SP7–SP12):

```
(* SP7: FAULT is Latched *)
FaultIsLatched ==
    mode_prev = "FAULT" => mode \in {"FAULT", "INERT"}

(* SP8: No Direct INERT to RUN *)
NoDirectINERTtoRUN ==
    mode_prev = "INERT"
        => mode \in {"INERT", "BOOT", "FAULT"}

(* SP9: BOOT Promotion Valid *)
BOOTPromotionValid ==
    (mode = "RUN" /\ mode_prev = "BOOT")
        => consecutive_ok >= N_PROMOTE

(* SP10: BOOT Timeout Bounded *)
BOOTTimeoutBounded ==
    mode = "BOOT"
        => (tick - boot_start) <= BOOT_TIMEOUT
                + DWELL_TICKS + 1

(* SP11: DENY Produces Zero Actuation *)
DenyProducesZeroActuation ==
    decision \in {"DENY", "HALT"}
        => actuation = "ZERO"

(* SP12: Chain Integrity Preservation *)
ChainIntegrityInvariant ==
    chain_intact = TRUE
```

B Lean 4 Theorem Statements

The following are the 20 theorem statements from `KevrosCorrect.lean`.

Core Invariant and Safety (14 theorems):

```
-- Core invariant
theorem inv_holds (nP bT dT : Nat) (s : State)
    (hR : Reachable nP bT dT s) : Inv s nP bT dT

theorem safety_holds (nP bT dT : Nat) (s : State)
    (hR : Reachable nP bT dT s) : SafetyInv s nP bT dT

-- Safety properties SP1-SP12
theorem SP1  ... : s.actuation = .applied -> s.tokenVerified = true
theorem SP2  ... : (phase in {evidence,token,respond})
    -> s.requestHmacOk = false -> s.decision = .deny
theorem SP3  ... : (phase in {token,respond})
    -> s.evidenceWritten = false
    -> s.writeSucceeded = false -> s.decision = .deny
theorem SP4  ... : s.tokenIssued = true -> s.evidenceWritten = true
theorem SP5  ... : s.actuation = .applied -> s.tokenIssued = true
theorem SP6  ... : (phase in {evidence,token,respond})
    -> (decision in {allow,clamp}) -> s.mode = .run
theorem SP7  ... : s.modePrev = .fault
    -> s.mode = .fault or s.mode = .inert
theorem SP8  ... : s.modePrev = .inert -> s.mode != .run
```

```
theorem SP9  ... : s.mode = .run -> s.modePrev = .boot
    -> s.consecutiveOk >= nP
theorem SP10 ... : s.mode = .boot
    -> s.tick - s.bootStart <= bT + dT + 1
theorem SP11 ... : (decision in {deny,halt}) -> s.actuation = .zero
theorem SP12 ... : s.chainIntact = true
```

Liveness (6 theorems):

```
theorem pipeline_progress ... :
    exists s', Next s s' and phaseDist s'.phase
        < phaseDist s.phase
theorem LP4_PipelineCompletes ... :
    exists s', NextStar s s' and s'.phase = .idle
theorem LP3_BootExits ... :
    exists s', Next s s' and s'.mode != .boot
theorem LP2_RunExits ... :
    exists s', Next s s' and s'.mode != .run
theorem LP1_BootProgress ... :
    exists s', Next s s' and (s'.mode = .run
        or (s'.mode = .boot
        and s'.consecutiveOk = s.consecutiveOk + 1))
theorem LP1_InertToBoot ... :
    exists s', Next s s' and s'.mode = .boot
```

C Verification Coverage Matrix

Table 7 shows which safety and liveness properties are verified by each layer. All 12 safety properties are verified by TLA+, Kani, Verus, and Lean 4. Liveness properties (LP1–LP4) are verified by TLA+/TLC (temporal logic under weak fairness) and Lean 4 (structural bounded progress via well-founded induction). Liveness properties are not applicable to Kani, Verus, RAC, or Golden Vectors, which verify state invariants rather than temporal properties.

Table 7: Safety and liveness property verification coverage by layer.

Property	**TLA+**	**Kani**	**Verus**	**RAC**	**GV**	**Lean 4**
SP1	✓	✓	✓	✓	✓	✓
SP2	✓	✓	✓	✓	✓	✓
SP3	✓	✓	✓	✓	✓	✓
SP4	✓	✓	✓	✓	✓	✓
SP5	✓	✓	✓	✓	✓	✓
SP6	✓	✓	✓	✓	✓	✓
SP7	✓	✓	✓	✓	–	✓
SP8	✓	✓	✓	✓	–	✓
SP9	✓	✓	✓	✓	–	✓
SP10	✓	✓	✓	✓	–	✓
SP11	✓	✓	✓	✓	✓	✓
SP12	✓	✓	✓	✓	✓	✓
LP1	✓	–	–	–	–	✓
LP2	✓	–	–	–	–	✓
LP3	✓	–	–	–	–	✓
LP4	✓	–	–	–	–	✓