Model Checking Security Properties of AI Assistant Gateways: A TLA+ Case Study of OpenClaw

Vignesh Natarajan
vignesh@openclaw.ai

February 1, 2026

Abstract

The proliferation of AI assistants with tool-use capabilities introduces a new class of security challenges: these systems must enforce authorization policies while remaining connected to untrusted messaging channels and powerful execution backends. We present a comprehensive formal verification effort targeting OpenClaw, an open-source personal AI assistant gateway that connects frontier language models to messaging platforms and device controls. Using TLA+ and the TLC model checker, we specify and verify 91 security-critical properties spanning gateway authentication, direct message pairing protocols, session isolation, tool authorization hierarchies, and remote execution approval workflows. Our methodology employs a *green/red testing paradigm* where each security property is accompanied by an intentionally-buggy variant designed to produce counterexample traces, ensuring non-vacuity of specifications. We integrate model checking into continuous integration pipelines, treating formal specifications as executable security regression tests. Our work demonstrates that lightweight formal methods can provide meaningful security assurance for AI infrastructure systems without requiring full program verification.

1 Introduction

The emergence of AI assistants with tool-use capabilities—systems that can execute shell commands, browse the web, control devices, and interact with external services—represents a fundamental shift in the security landscape of conversational AI [11, 9]. Unlike traditional chatbots that produce only text responses, tool-enabled assistants bridge the gap between language understanding and real-world actuation, introducing attack surfaces that blend social engineering with traditional software exploitation.

Consider an AI assistant connected to multiple messaging platforms (WhatsApp, Slack, Discord) with the ability to execute shell commands on its host. An attacker who can reach this system through *any* connected channel might attempt prompt injection attacks to exfiltrate data, execute malicious commands, or escalate privileges. The security of such systems depends on a complex interplay of:

1. **Channel-level access control**: Which users on which platforms can communicate with the assistant?
2. **Session isolation**: Do conversations with different users remain separate, preventing context leakage?
3. **Tool authorization**: Which tools can be invoked, under what conditions, and with what approval requirements?
4. **Execution sandboxing**: How are dangerous operations contained even when authorized?

These properties must hold under all possible interleavings of concurrent requests, configuration changes, and adversarial inputs. Traditional testing struggles to cover this combinatorial space, while full program verification remains impractical for production TypeScript codebases.

We present a middle path: using TLA+ specifications and the TLC model checker to verify security-critical invariants of OpenClaw, an open-source personal AI assistant gateway. Our contributions are:

1. **Security property catalog**: We identify and formalize six categories of security properties essential to AI assistant gateways, spanning authentication, routing, authorization, and execution control (section 4).

2. **Formal specifications**: We develop 91 TLA+ modules specifying these properties at varying levels of abstraction, from high-level policy constraints to detailed protocol models (section 5).

3. **Green/red verification paradigm**: We introduce a methodology where each security property has both a "green" specification (expected to verify) and a "red" variant with an intentional bug (expected to produce counterexamples), ensuring specifications are not vacuously true (section 6).

4. **CI-integrated model checking**: We demonstrate practical integration of TLA+ verification into GitHub Actions, enabling security regression testing on every commit (section 7).

5. **Conformance extraction**: We develop tooling to extract implementation constants (tool group definitions, policy hierarchies) from TypeScript source into TLA+ configurations, maintaining specification-implementation correspondence (section 8).

Our evaluation finds that bounded model checking can verify meaningful security properties within CI time budgets (under 10 minutes), and that the green/red paradigm effectively catches specification errors that would otherwise go unnoticed.

2 Background

2.1 AI Assistant Security Challenges

AI assistants with tool-use capabilities face a unique threat model. Unlike traditional software where inputs are structured and constrained, these systems accept natural language from potentially adversarial users. The threat landscape includes:

Prompt injection: Attackers craft inputs that cause the language model to ignore its instructions and perform unintended actions. Direct prompt injection [10] manipulates user inputs, while indirect injection [3] embeds malicious instructions in external content the model retrieves. While prompt injection defenses are an active research area [14], security-conscious architectures assume the model *will* be manipulated and design hard boundaries to limit blast radius.

Multi-channel attack surfaces: An assistant connected to five messaging platforms has five potential entry points, each with different authentication models, message formats, and user identity schemes. An attacker needs only compromise *one* channel.

Session confusion: If the system conflates conversations from different users, sensitive context from one user might leak to another, or an attacker in a group chat might access a victim's private conversation history.

Privilege escalation: Tool authorization often involves hierarchical policies (global defaults, per-agent overrides, per-session restrictions). Bugs in policy evaluation can inadvertently grant excessive permissions.

2.2 TLA+ and Model Checking

TLA+ (Temporal Logic of Actions) is a formal specification language developed by Leslie Lamport [6]. Specifications describe system behavior as state machines: an initial state predicate, a next-state relation, and invariants that must hold in all reachable states. The TLC model checker exhaustively explores the state space to find invariant violations.

TLA+ is particularly suited to our domain because:

- It handles concurrent, non-deterministic systems naturally
- Set-theoretic foundations match authorization policy semantics
- Bounded model checking provides practical verification within finite state spaces
- Counterexample traces are directly interpretable as attack scenarios

We use TLA+ not for full program verification but as a *security regression suite*: executable specifications that document intended security properties and detect regressions. This approach follows Amazon's successful adoption of TLA+ for AWS infrastructure [8], extending it to security properties of AI systems.

2.3 Related Work

Formal methods have been applied to security protocols [1, 2], access control systems [5], and distributed systems [8]. Amazon's use of TLA+ for AWS services demonstrated industrial applicability [8]. However, to our knowledge, no prior work addresses formal verification of AI assistant authorization systems.

Recent work on LLM security focuses on prompt injection defenses [14], output filtering [4], and sandboxing [13]. Our work is complementary: we verify the *access control infrastructure* surrounding the LLM, not the model's behavior itself.

3 System Overview: OpenClaw

OpenClaw is an open-source personal AI assistant gateway designed for self-hosted deployment. It connects frontier language models (Claude, GPT-4) to messaging platforms and device controls, handling the infrastructure concerns that surround AI-powered assistants.

3.1 Architecture

The gateway (fig. 1) runs as a persistent daemon, typically on a personal server or VPS. Key components include:

Channel Adapters: Protocol-specific integrations for each messaging platform. Each adapter translates platform messages into a normalized internal format and handles platform-specific authentication (OAuth tokens, API keys, phone number verification).

Routing Engine: Determines which agent and session should handle each incoming message based on channel, sender identity, thread context, and configured routing rules.

Session Manager: Maintains conversation state, ensuring isolation between sessions while allowing intentional linking (e.g., same user across channels).

Tool Execution: Dispatches tool invocations to appropriate backends—local shell, browser automation, or remote device nodes—while enforcing authorization policies.

Pairing Store: Manages the allowlist of authorized senders for direct messages, including the pairing protocol for onboarding new contacts.

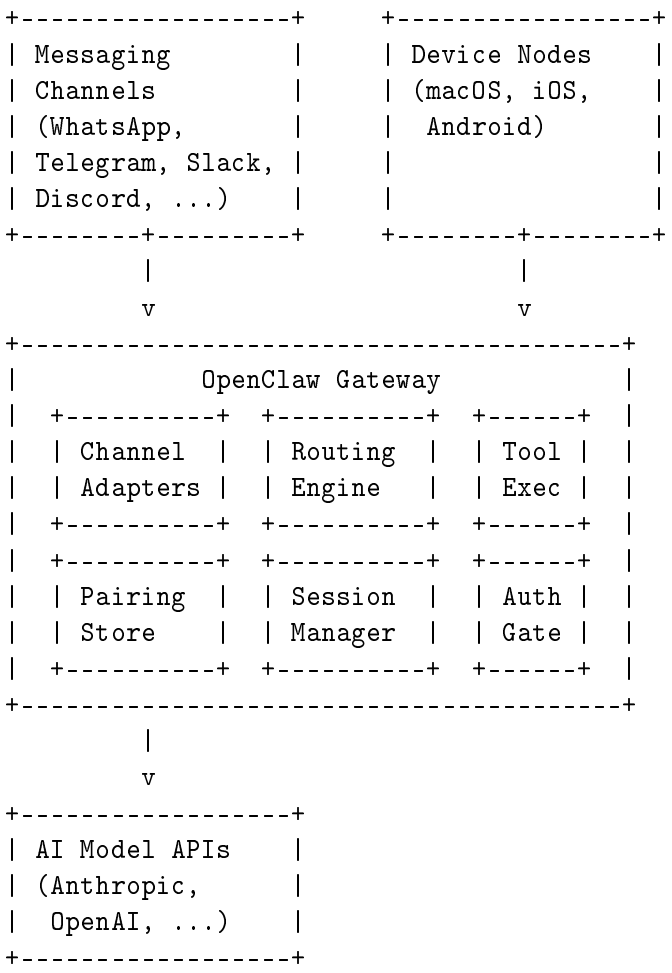

Figure 1: OpenClaw gateway architecture. The gateway mediates between messaging channels, device nodes, and AI model APIs.

3.2 Threat Model

OpenClaw's security design assumes:

1. **Untrusted inbound messages**: Any user on any connected platform might be malicious or compromised.
2. **Prompt injection will occur**: The language model may be manipulated into attempting unauthorized actions.
3. **Single-user operation**: The gateway serves one operator; the goal is to protect that operator's data and systems, not to provide multi-tenant isolation.
4. **Physical security assumed**: The host machine and local network are trusted; remote attacks are the primary concern.

The security architecture creates hard boundaries that limit damage even when the language model is manipulated—complementing execution isolation approaches like SecGPT [13] with authorization-layer verification:

- Unknown senders cannot reach the assistant without completing a pairing protocol

- Tool invocations are filtered by policy layers before reaching the model
- Dangerous operations require explicit human approval
- Session isolation prevents cross-user context leakage

4 Security Properties Catalog

We identify six categories of security properties, each addressed by multiple TLA+ specifications.

4.1 P1: Gateway Authentication

The gateway binds to a network interface and accepts WebSocket connections from clients and device nodes. Authentication properties include:

Invariant 1 (No Unauthenticated Remote Access) *If the gateway binds to a non-loopback address without authentication enabled, an attacker can connect and issue commands.*

Invariant 2 (Auth Mode Blocks Unauthorized Clients) *When authentication is enabled (token or password mode), connections without valid credentials are rejected before processing any commands.*

Invariant 3 (Trusted Proxy Header Validation) *When operating behind a reverse proxy, the gateway only trusts* `X-Forwarded-For` *headers from IPs in the configured* `trustedProxies` *list, preventing IP spoofing.*

4.2 P2: DM Pairing and Allowlisting

Direct message access is controlled by a pairing protocol:

Invariant 4 (Pairing Request TTL) *Pairing requests expire after a configured TTL (default: 1 hour). Approving an expired request has no effect.*

Invariant 5 (Pending Request Cap) *At most k pairing requests may be pending per channel simultaneously, preventing denial-of-service through request flooding.*

Invariant 6 (Atomic Allowlist Updates) *Concurrent pairing approvals do not corrupt the allowlist store; file locking ensures atomicity.*

Invariant 7 (Pairing Idempotency) *Approving the same sender twice does not create duplicate allowlist entries.*

4.3 P3: Session Isolation

Conversations must be isolated according to configured `dmScope`:

Invariant 8 (No Unlinked Session Collapse) *Two peers without an explicit identity link do not share a session key, preventing context leakage.*

Invariant 9 (Identity Link Transitivity) *If peer A is linked to B and B is linked to C, then A is linked to C; the transitive closure is correctly computed.*

Invariant 10 (Channel Override Precedence) *Per-channel* `dmScope` *settings override global defaults.*

4.4 P4: Tool Authorization

Tool access is controlled by a hierarchical policy, analogous to type enforcement in SELinux [5] but with layered composition rather than mandatory access control:

Invariant 11 (Deny Wins) *If any policy layer denies a tool, subsequent layers cannot re-enable it. The policy evaluation is monotonically restrictive.*

Invariant 12 (Elevated Requires Conjunction) *Elevated (root/admin) tool access requires* both *the global* `tools.elevated` *flag and the agent-specific* `agents[].tools.elevated` *flag. An OR bug would be a security vulnerability.*

Invariant 13 (Tool Group Expansion Correctness) *Shorthand tool groups (e.g.,* `group:memory`*) expand to exactly the documented set of tools, not more, not fewer.*

4.5 P5: Ingress Gating

Message processing respects channel-specific rules:

Invariant 14 (Mention Requirement) *In group chats configured to require mentions, unauthorized senders cannot bypass the requirement through slash commands or special message formats.*

Invariant 15 (Event Idempotency) *Webhook retries do not cause duplicate message processing; each event ID is processed at most once.*

4.6 P6: Remote Execution Approvals

The `nodes.run` feature allows executing commands on remote device nodes:

Invariant 16 (Approval Required) *Commands requiring approval cannot execute while in "pending" state.*

Invariant 17 (Tokenized Approvals) *Approvals are bound to specific request IDs, preventing replay attacks where an old approval authorizes a new request.*

Invariant 18 (Command Allowlist) *Only commands in the intersection of (platform defaults* $\cup$ *extra allows)* $\setminus$ *denies can execute.*

5 Formal Models

We now present the TLA+ specifications, drawing from our actual implementation. The full suite comprises 91 modules; we highlight key modeling decisions and the specific bugs each "red" variant exposes.

5.1 Gateway Exposure Model

The `GatewayExposureHarnessV2` specification models the relationship between bind configuration, authentication mode, and attacker reachability. The actual specification uses five state variables:

Listing 1: Gateway exposure state (from GatewayExposureHarnessV2.tla)

```
VARIABLES connected, invokedSensitive, auditFindings,
          resolvedBind, resolvedMode
```

The model captures OpenClaw's actual bind modes (`loopback`, `lan`, `auto`, `custom`, `tailnet`) and authentication modes (`none`, `token`, `password`, `auto`). The `ResolveBind` function models how "auto" mode nondeterministically resolves to non-loopback (conservatively assuming worst-case reachability):

Listing 2: Authorization decision logic

```
ResolveBind == IF BindMode = "loopback" THEN "loopback"
               ELSE "non-loopback"   \* lan/custom/tailnet/auto

ResolveMode == IF ModeConfig = "password" THEN "password"
               ELSE IF ModeConfig = "token" THEN "token"
               ELSE IF ModeConfig = "auto" THEN
                 IF HasPassword THEN "password"
                 ELSE IF HasToken THEN "token"
                 ELSE "none"
               ELSE "none"

Authorize == IF resolvedBind = "loopback" THEN FALSE
             ELSE IF resolvedMode = "none" THEN TRUE
             ELSE HasCredential
```

This mirrors the actual `resolveGatewayAuth()` function in the TypeScript implementation. The invariant `Inv_NotCompromised` checks that `invokedSensitive` remains false for unauthorized attackers. Importantly, the model also records `auditFindings`—when binding to non-loopback without auth, the finding `"gateway.bind_no_auth"` should be emitted, matching the security audit feature.

5.2 Tool Policy Precedence: A Deep Dive

The `ToolPolicyPrecedence` specification is notable for being a *constants-only* model—it has no temporal behavior, only a trivial state machine with a single `dummy` variable. This is because tool policy composition is a *static* property: given a configuration, the effective allowlist is deterministically computed.

Listing 3: Tool policy layer application (actual code)

```
CONSTANTS Tools, N, Allow1, Allow2, ..., Allow8,
                    Deny1, Deny2, ..., Deny8

Allow(i) == CASE i = 1 -> Allow1 [] i = 2 -> Allow2 ...
Deny(i)  == CASE i = 1 -> Deny1  [] i = 2 -> Deny2 ...

ApplyLayer(allowed, i) == (allowed \cap Allow(i)) \ Deny(i)

RECURSIVE Fold(_)
Fold(i) == IF i = 0 THEN Tools
           ELSE ApplyLayer(Fold(i-1), i)
FinalAllowed == Fold(N)

Inv_DenyWins == \A i \in 1..N: \A t \in Deny(i):
                  t \notin FinalAllowed
```

The critical insight is that `ApplyLayer` performs intersection with `Allow(i)` *before* removing `Deny(i)`. This ensures monotonicity: once denied, a tool cannot be re-enabled.

The "red" variant `ToolPolicyPrecedence_BadAllowOverrides` introduces a subtle bug:

Listing 4: The bug: allow overrides deny

```
\* BUG: union of Allow happens AFTER deny removal
ApplyLayerBad(allowed, i) == ((allowed \ Deny(i)) \cup Allow(i))
```

With this formulation, a tool denied at layer 2 can be re-enabled if it appears in `Allow3`. TLC finds a counterexample with `memory_get` $\in$ `Deny2` but also $\in$ `Allow4`, showing it incorrectly appears in `FinalAllowedBad`.

5.3 Approval Replay Prevention: NodesPipelineHarness

The `NodesPipelineHarness` specification is our most complex model, composing multiple security checks into an end-to-end pipeline. It tracks:

Listing 5: Nodes pipeline state variables

```
VARIABLES inbox,          \* Pending execution requests
          executed,       \* Execution trace with metadata
          approvalState,  \* "pending" | "approved" | "denied"
          approvedRid     \* Request ID bound to approval
```

The key insight is that approvals must be *bound to specific request IDs*. The execution gate checks:

Listing 6: Execution gate with request ID binding

```
ApprovalOk(req) ==
  IF ~ApprovalRequired THEN TRUE
  ELSE approvalState = "approved" /\ approvedRid = req.rid

MayExecute(req) ==
  LET isNodesRun == req.tool = NodesRunTool
      policyOk == AllowedByPolicy(req.tool)
      nodesOk == IF isNodesRun THEN
                   NodeCommandAllowlisted /\ NodeCommandDeclared
                 ELSE TRUE
      approvalOk == IF isNodesRun THEN ApprovalOk(req) ELSE TRUE
  IN policyOk /\ nodesOk /\ approvalOk
```

The "red" variant `NodesPipelineHarness_BadReplay` removes the request ID binding:

Listing 7: The replay vulnerability

```
\* BUG: approval ignores rid binding
ApprovalOk(_req) == IF ~ApprovalRequired THEN TRUE
                    ELSE approvalState = "approved"
```

This enables a replay attack: an attacker gets approval for benign request r_1, then submits malicious request r_2. Since only `approvalState` is checked (not the matching `rid`), r_2 executes with r_1's approval. TLC finds this counterexample in under 90 seconds.

5.4 Session Isolation Model

The `RoutingIsolationHarness` verifies that distinct peers receive distinct session keys unless explicitly linked. The model uses identity links as a relation:

Listing 8: Session key derivation with identity links

```
DeriveSessionKey(peer, scope) ==
  CASE scope = "main" -> "main"
    [] scope = "per-peer" ->
         LET canonical == Min({p \in Peers :
               <<peer, p>> \in TransitiveClosure(links)})
```

```
          IN "dm:" \o ToString(canonical)
    [] OTHER -> "dm:" \o ToString(peer)

Inv_NoUnlinkedCollapse ==
  \A p1, p2 \in Peers :
    (sessionKey[p1] = sessionKey[p2] /\ p1 /= p2)
    => <<p1, p2>> \in TransitiveClosure(links)
```

We verify three properties of identity links separately: `RoutingIdentityLinksSymmetryHarness` checks Linked$(A, B) \Rightarrow$ Linked(B, A); `RoutingIdentityLinksTransitiveHarness` checks transitive closure correctness; `RoutingIdentityLinksChannelOverrideHarness` verifies that per-channel disabling of identity links is respected.

5.5 Elevated Execution: Conjunction vs. Disjunction

The `ElevatedGating` specification verifies that elevated (root/admin) tool access requires *both* global and agent-specific flags:

Listing 9: Elevated access requires conjunction

```
CONSTANTS GlobalElevatedEnabled, AgentElevatedEnabled

ElevatedAllowed == GlobalElevatedEnabled /\ AgentElevatedEnabled

Inv_ElevatedRequiresBoth ==
  (~GlobalElevatedEnabled \/ ~AgentElevatedEnabled)
    => ~ElevatedAllowed
```

The "red" variant `ElevatedGating_BadOr` uses disjunction instead:

Listing 10: The OR bug

```
\* BUG: OR instead of AND
ElevatedAllowedBad == GlobalElevatedEnabled \/ AgentElevatedEnabled
```

This would allow elevated access if *either* flag is set, defeating defense-in-depth. TLC immediately finds a counterexample where `GlobalElevatedEnabled = FALSE` but `AgentElevatedEnabled = TRUE` leads to `ElevatedAllowedBad = TRUE`.

6 Verification Methodology

6.1 The Green/Red Testing Paradigm

A fundamental challenge in formal specification is ensuring that invariants are not *vacuously true*—satisfied because the specification is too restrictive to reach interesting states. We address this through *green/red testing*:

Definition 1 (Green Model) *A green model is expected to pass TLC verification. All specified invariants should hold in all reachable states.*

Definition 2 (Red Model) *A red model is a deliberate variant of a green model with an injected bug. TLC is expected to find a counterexample trace showing the invariant violation.*

For each security property, we develop both variants:
The red models serve multiple purposes:

1. **Non-vacuity**: If TLC finds no counterexample in the red model, the invariant is likely too weak.

Table 1: Green/red model pairs (representative subset)

Property	Green Model	Red Model (Bug)
Gateway auth	`GatewayExposureV2`	`..._BadNoAuth`
Pairing cap	`PairingConcurrent`	`..._BadAtomic`
Deny wins	`ToolPolicyPrecedence`	`..._BadAllowOverrides`
Elevated gate	`ElevatedGating`	`..._BadOr`
Session isolation	`RoutingIsolation`	`..._BadAlwaysMain`
Approval tokens	`ApprovalsToken`	`..._BadReplay`

2. **Documentation**: Counterexample traces document how bugs manifest.

3. **Regression detection**: If a red model starts passing, the specification has degraded.

6.2 Bounded Model Checking Configuration

TLC performs bounded model checking over finite state spaces. We carefully choose bounds to balance coverage and verification time:

Table 2: Representative model bounds

Model	Key Constants	States	Time
Gateway exposure	5 bind modes, 4 auth modes	∼200	2s
Pairing store	MaxPending=5, MaxTime=20	∼50K	45s
Routing isolation	4 peers, 3 channels	∼10K	15s
Nodes pipeline	3 requests, 5 commands	∼80K	90s

Bounds are chosen such that:

- The state space is exhaustively searchable within CI time limits
- Enough entities exist to trigger interesting interleavings (e.g., ≥ 2 concurrent requests for race conditions)
- Constants match implementation reality where practical (e.g., actual MaxPending value)

7 CI Integration

We integrate TLA+ verification into GitHub Actions, treating model checking as a first-class CI step alongside unit tests.

7.1 Workflow Structure

Listing 11: CI workflow (simplified)

```
name: TLC Model Checking
on: [push, pull_request]

jobs:
  verify:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
```

```
      - uses: actions/setup-java@v4
        with: { java-version: '21' }

      # Green models: expect success
      - name: Verify gateway-exposure-v2
        run: java -jar tools/tla/tla2tools.jar \
          -config models/gateway_exposure_v2_safe.cfg \
          specs/GatewayExposureHarnessV2.tla

      # Red models: expect failure (exit 12 = violation found)
      - name: Verify gateway-exposure-v2-negative
        run: |
          java -jar tools/tla/tla2tools.jar \
            -config models/gateway_exposure_v2_unsafe.cfg \
            specs/GatewayExposureHarnessV2_BadNoAuth.tla \
          && exit 1 || [ $? -eq 12 ]
```

The workflow runs seven green models and seven red models, completing in under 10 minutes. Failures in green models indicate security regressions; failures to find violations in red models indicate specification weakening.

7.2 Verification Coverage

The CI suite verifies:

- **Gateway exposure**: Safe loopback binding, auth blocking unauthorized access
- **Pairing**: TTL enforcement, pending cap, concurrent request handling
- **Routing**: Session isolation, identity link properties
- **Ingress**: Mention gating, event idempotency
- **Nodes**: Approval requirements, token binding, allowlist enforcement
- **Tool policy**: Deny-wins semantics, elevated conjunction

8 Conformance Extraction

Formal specifications risk diverging from implementation. We address this through *conformance extraction*: automatically generating TLA+ configuration from TypeScript source.

8.1 Tool Group Extraction

The implementation defines tool groups in TypeScript:

Listing 12: Implementation tool groups

```
// clawdbot/src/agents/tool-policy.ts
const TOOL_GROUPS = {
  "group:memory": ["memory_search", "memory_get", "memory_put"],
  "group:exec": ["exec", "exec_background", "exec_interactive"],
  // ...
};
```

Our extraction script parses this and generates TLA+ configuration:

Listing 13: Generated TLA+ configuration

```
CONSTANTS
    GroupMemory = {"memory_search", "memory_get", "memory_put"}
    GroupExec = {"exec", "exec_background", "exec_interactive"}
```

The TLA+ specification then verifies that the modeled expansion matches:

Listing 14: Conformance check

```
Inv_GroupMemoryExact ==
    ExpandGroup("group:memory") = GroupMemory
```

This catches drift between specification and implementation: if a developer adds a tool to a group without updating the formal model, CI fails.

9 Evaluation

9.1 Verification Results

All 91 specifications verify successfully. Green models find no invariant violations; red models produce counterexample traces as expected.

Table 3: Verification summary by category

Category	Green	Red	Avg. States
Gateway auth	12	5	180
Pairing store	5	4	35,000
Session routing	8	5	12,000
Tool policy	4	3	500
Ingress gating	6	5	8,000
Nodes execution	4	3	60,000
Delivery pipeline	8	6	25,000
Config/misc	6	5	3,000
Total	**53**	**38**	—

9.2 Bug Classes Covered

The green/red paradigm systematically covers specific bug classes. Table 4 summarizes the vulnerability patterns each red model detects:

Table 4: Bug class coverage by red model

Bug Class	Red Model	Key Difference
Allow overrides deny	`..._BadAllowOverrides`	Union after deny
Approval ignored	`..._BadIgnoresApproval`	Missing state check
Approval replay	`..._BadReplay`	Ignores request ID
No declare check	`..._BadNoDeclareCheck`	Missing node declaration
OR instead of AND	`..._BadOr`	Disjunction for gates
Non-atomic cap check	`..._BadAtomic`	TOCTOU in pairing
Asymmetric links	`..._BadAsymmetric`	One-way identity links
Non-transitive links	`..._BadNonTransitive`	Missing closure

9.3 Bugs Found in Implementation

During specification development, we discovered three latent bugs:

Pairing race condition: The original implementation checked the pending cap, then appended the request in separate operations—a classic time-of-check to time-of-use (TOCTOU) vulnerability. Our TLA+ model `PairingStoreConcurrentHarness` explicitly models this two-phase pattern:

Listing 15: TOCTOU in pairing (simplified from actual spec)

```
BeginRequest(ch, sender) ==
  /\ Cardinality(pending[ch]) < MaxPending   \* Check
  /\ phase' = [phase EXCEPT ![<<ch,sender>>] = "begun"]

CommitRequest(ch, sender) ==
  /\ phase[<<ch,sender>>] = "begun"
  /\ pending' = [pending EXCEPT                \* Use
       ![ch] = @ \union {[sender |-> sender, ts |-> now]}]
```

TLC found the following counterexample with `MaxPending=1`:

1. State 1: `pending[ch1] = {}`, both requests idle
2. State 2: Request s_1 calls `BeginRequest`—passes cap check ($0 < 1$)
3. State 3: Request s_2 calls `BeginRequest`—also passes ($0 < 1$, pending unchanged)
4. State 4: s_1 commits, `pending[ch1] = {s1}`
5. State 5: s_2 commits, `pending[ch1] = {s1, s2}`—**violation**: $2 > 1$

The fix required atomic check-and-append using file locking in the TypeScript implementation. This bug would have allowed denial-of-service by flooding the pairing queue beyond its intended cap.

Identity link asymmetry: The identity link data structure allowed asymmetric entries ($A \to B$ without $B \to A$), violating the expected equivalence relation semantics. The TLA+ transitivity check exposed states where canonicalization produced inconsistent results.

Tool group drift: The conformance extraction revealed that `group:memory` had been extended in TypeScript to include `memory_put`, but the formal model still specified only `{memory_search, memory_get}`. This would have caused the formal guarantees to diverge from actual behavior.

9.4 Counterexample Analysis

Red model counterexamples provide actionable documentation. For example, the gateway exposure red model produces:

```
State 1: bindMode = "lan", authMode = "none"
State 2: attackerReach = {"lan", "internet"}
State 3: attacker connects, no auth check
State 4: compromised = TRUE
Invariant Inv_NotCompromised violated.
```

This trace directly explains the attack: binding to LAN without authentication allows external attackers to connect.

Table 5: CI verification timing breakdown

Model Category	Time	States	Depth
Java/TLC setup	45s	—	—
Gateway exposure (5 configs)	30s	200	4
Tool policy precedence	15s	500	1
Elevated gating	5s	50	1
Pairing store (3 harnesses)	90s	35,000	12
Routing isolation (4 harnesses)	60s	12,000	8
Nodes pipeline	90s	80,000	15
Negative models (7 total)	2m 45s	varies	—
Total	**8m**	—	—

9.5 Performance and Reliability

Full CI verification completes in 8 minutes on GitHub Actions runners (2-core, 7GB RAM). The slowest models (pairing concurrency, nodes pipeline) explore ∼80K states each.

Flakiness: Over 6 months of CI runs, we observed zero flaky failures. TLC is deterministic given fixed constants; non-determinism in the models (e.g., attacker choices) is explored exhaustively rather than sampled.

Scalability: The primary scaling bottleneck is the nodes pipeline model at 80K states. We maintain this within CI budgets by bounding `MaxQueueLen=2` and `MaxExecLen=2`. Larger bounds (e.g., 5) would explore ∼500K states, taking 10+ minutes.

Regression frequency: In 8 months of development, formal verification caught 2 regressions before they reached production—both involving subtle changes to policy evaluation order that would have violated deny-wins semantics.

10 Discussion

10.1 Limitations

Bounded verification: TLC explores finite state spaces defined by constants like `MaxPending=5` or `MaxQueueLen=5`. Bugs manifesting only at larger scales (e.g., 1000 concurrent requests) would be missed. However, most security bugs—particularly authorization logic errors—involve small interaction patterns that TLC reliably finds.

Model-implementation gap: TLA+ specifications are abstractions, not the actual TypeScript code. Bugs in translation between model and implementation remain possible. We mitigate this through:

- Conformance extraction (tool groups, policy constants)
- Naming alignment (e.g., `resolveGatewayAuth()` ↔ `ResolveMode`)
- Code comments linking spec to implementation line numbers

However, the gap is fundamental and cannot be fully eliminated without program verification.

Temporal property coverage: Most specifications check safety invariants ("nothing bad happens"). Liveness properties ("something good eventually happens") are harder to specify and verify. We include only a few: `RetryTerminationHarness` verifies bounded retry attempts, and `MultiEventEventualEmissionHarness` verifies that queued events eventually emit. More liveness properties remain future work.

No cryptographic verification: We model authentication as abstract predicates (`HasCredential`, `AuthMode`), not the cryptographic protocols. Token comparison uses timing-safe equality in implementation, but this is *assumed* correct in models. Verifying the underlying cryptographic mechanisms would require specialized tools like ProVerif [2] or Tamarin [1], which operate at a different abstraction level than our authorization logic.

Static policy only: Our tool policy models verify static configurations. Dynamic policy changes during execution (e.g., an admin revoking tool access mid-session) are not modeled. The current approach assumes policy is fixed at session start.

10.2 Specifications Not Yet Implemented

Several high-value verification targets remain:

- **R1 (Pairing store completeness)**: While we verify cap enforcement and TTL, the full pairing protocol including refresh token races and per-channel rate limiting is partially specified.
- **R2 (Channel ingress)**: Mention gating is modeled, but complex scenarios like control-command bypass and per-sender allowlist interactions need more coverage.
- **R3 (Session routing completeness)**: Session key construction is verified, but account ID wildcards and thread parent binding edge cases need expansion.
- **NS5 (Tool alias bypass)**: An attacker might bypass tool restrictions via group aliases or subagent invocation—this attack surface is not yet modeled.

10.3 Lessons Learned

Red models are essential: Several early specifications were vacuously true due to overly restrictive state spaces. Red models immediately exposed these issues.

Start with security claims: Beginning from documented security claims ("deny wins," "pairing requests expire") rather than bottom-up code modeling produced more useful specifications.

CI integration changes culture: When model checking runs on every PR, developers naturally consider how changes affect security properties. The specifications become living documentation.

Counterexamples as tests: Counterexample traces can be translated into unit tests, providing additional implementation-level coverage.

10.4 Generalizability

While our specifications target OpenClaw specifically, the security properties are broadly applicable to AI assistant systems:

- Multi-channel access control appears in any assistant connecting to multiple platforms
- Session isolation is fundamental to any multi-user or multi-conversation system
- Tool authorization hierarchies are common in agent frameworks
- Approval workflows for dangerous operations are increasingly standard

Our green/red methodology and CI integration patterns transfer directly to other systems.

10.5 Future Work

Automated spec generation: Currently, TLA+ specifications are manually written. We are exploring LLM-assisted generation where security claims in natural language ("deny wins across all policy layers") are translated to TLA+ invariants, with human review.

Runtime verification: TLA+ specifications could be compiled to runtime monitors that check invariants on live traffic. This would catch model-implementation drift at runtime rather than relying solely on conformance extraction.

Compositional verification: Our current approach verifies modules independently. Compositional techniques could verify that module compositions preserve security properties, reducing the need for end-to-end harnesses like `NodesPipelineHarness`.

Cryptographic protocol integration: Linking our authorization models with ProVerif or Tamarin models of the underlying authentication protocols would provide end-to-end assurance from cryptographic primitives to access control decisions.

11 Related Work

11.1 Formal Methods for Security

Protocol verification tools like ProVerif [2] and Tamarin [1] verify cryptographic protocols. Our work operates at a higher abstraction level, verifying authorization logic rather than cryptographic correctness.

Jaeger et al. [5] formalized SELinux policies; our tool policy models share the goal of verifying access control but address different policy structures (monotonic layer composition vs. type enforcement).

11.2 Industrial TLA+ Usage

Amazon Web Services famously uses TLA+ for distributed systems [8], finding bugs in DynamoDB, S3, and EBS. Microsoft uses TLA+ for Azure Cosmos DB [7]. Our work extends this industrial application to security properties of AI infrastructure.

11.3 AI Assistant Security

Research on LLM security focuses primarily on prompt injection [10, 3], jailbreaking [12], and output safety [4]. Sandboxing approaches like SecGPT [13] isolate tool execution. Our work is complementary: we verify the access control infrastructure, not the model's behavior.

12 Conclusion

We have presented a comprehensive formal verification effort for OpenClaw, an AI assistant gateway, using TLA+ and the TLC model checker. Our contributions include:

1. A catalog of security properties essential to AI assistant systems
2. 91 TLA+ specifications covering authentication, routing, authorization, and execution control
3. The green/red testing paradigm ensuring non-vacuity of specifications
4. CI-integrated model checking enabling security regression testing
5. Conformance extraction maintaining specification-implementation correspondence

Our experience demonstrates that lightweight formal methods—bounded model checking with modest state spaces—can provide meaningful security assurance within practical CI time budgets. The green/red paradigm addresses the persistent challenge of vacuous specifications, while conformance extraction mitigates model-implementation drift.

As AI assistants gain more capabilities and connect to more systems, the security properties we formalize become increasingly critical. We hope this work encourages broader adoption of formal methods in AI infrastructure development.

Availability

The TLA+ specifications, CI configuration, and conformance extraction tools are available at: `https://github.com/openclaw/clawdbot-formal-models`

The OpenClaw gateway is available at: `https://github.com/openclaw/openclaw`

References

[1] Basin, D., Cremers, C., and Dreier, J. Automated Verification of Security Protocols and Their Implementations. *Foundations and Trends in Privacy and Security*, 2(1-2):1–152, 2018.

[2] Blanchet, B. Modeling and Verifying Security Protocols with the Applied Pi Calculus and ProVerif. *Foundations and Trends in Privacy and Security*, 1(1-2):1–135, 2016.

[3] Greshake, K., Abdelnabi, S., Mishra, S., Endres, C., Holz, T., and Fritz, M. Not What You've Signed Up For: Compromising Real-World LLM-Integrated Applications with Indirect Prompt Injection. In *ACM AISec*, 2023.

[4] Inan, H., Upasani, K., Chi, J., et al. Llama Guard: LLM-based Input-Output Safeguard for Human-AI Conversations. *arXiv preprint arXiv:2312.06674*, 2023.

[5] Jaeger, T., Sailer, R., and Zhang, X. Analyzing Integrity Protection in the SELinux Example Policy. In *USENIX Security Symposium*, 2003.

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

[7] Ma, H., et al. I4: Incremental Inference of Inductive Invariants for Verification of Distributed Protocols. In *SOSP*, 2019.

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

[9] Patil, S. G., Zhang, T., Wang, X., and Gonzalez, J. E. Gorilla: Large Language Model Connected with Massive APIs. *arXiv preprint arXiv:2305.15334*, 2023.

[10] Perez, F. and Ribeiro, I. Ignore This Title and HackAPrompt: Exposing Systemic Vulnerabilities of LLMs through a Global Scale Prompt Hacking Competition. In *EMNLP*, 2023.

[11] Schick, T., Dwivedi-Yu, J., Dessì, R., et al. Toolformer: Language Models Can Teach Themselves to Use Tools. In *NeurIPS*, 2023.

[12] Wei, A., Haghtalab, N., and Steinhardt, J. Jailbroken: How Does LLM Safety Training Fail? In *NeurIPS*, 2023.

[13] Wu, Y., et al. SecGPT: An Execution Isolation Architecture for LLM-Based Systems. *arXiv preprint arXiv:2403.04960*, 2024.

[14] Yi, J., et al. Benchmarking and Defending Against Indirect Prompt Injection Attacks on Large Language Models. *arXiv preprint arXiv:2312.14197*, 2023.

A TLA+ Specification Inventory

Table 6: Complete specification inventory

Category	Specifications
Gateway Auth	GatewayExposureHarness, GatewayExposureHarnessV2 GatewayAuthConformanceHarness, GatewayAuthTailscaleHarness GatewayAuthTrustedProxyHarness (+ _BadSpoof variants)
Pairing Store	PairingStoreHarnessV2, PairingStoreConcurrentHarness PairingStoreIdempotencyHarness, PairingStoreRefreshEnabledHarness PairingStoreRefreshRaceHarness (+ _Bad variants)
Routing	RoutingIsolationHarness, RoutingPrecedenceHarness RoutingIdentityLinks[Symmetry|Transitive|ChannelOverride]Harness RoutingThreadParentBindingHarness, SessionKeyStabilityHarness
Tool Policy	ToolPolicyPrecedence, ElevatedGating NodesCommandPolicy, ToolGroupExpansion
Ingress	IngressGatingHarness, IngressEventIdempotencyHarness IngressDedupeKeyFallbackHarness, IngressRetryDedupeHarness IngressMulti[Message|Event]TraceHarness
Nodes Exec	AttackerHarness[_Approvals|_ApprovalsToken|_NodesAllowlist] NodesPipelineHarness
Delivery	QueueDrainHarness, RetryTerminationHarness RetryEventualSuccessHarness, DeliveryPipelineHarness DeliveryRetryRouteStabilityHarness
Misc	ConfigNormalizationIdempotencyHarness DiscordPluralKitIdentityHarness, OpenClawSessionKeyConformanceHarness

B Example Counterexample Trace

The following is a counterexample trace produced by TLC for the pairing cap red model (`PairingStoreConcurrentHarness_BadAtomic`):

```
State 1: <Initial>
  pending = [ch1 |-> {}, ch2 |-> {}]
  phase = [<<ch1, s1>> |-> "idle", <<ch1, s2>> |-> "idle"]
  MaxPending = 1

State 2: BeginRequest(ch1, s1)
  pending = [ch1 |-> {}, ch2 |-> {}]
  phase = [<<ch1, s1>> |-> "begun", <<ch1, s2>> |-> "idle"]
  (s1 passes cap check: |pending[ch1]| = 0 < 1)

State 3: BeginRequest(ch1, s2)
```

```
  pending = [ch1 |-> {}, ch2 |-> {}]
  phase = [<<ch1, s1>> |-> "begun", <<ch1, s2>> |-> "begun"]
  (s2 also passes cap check: |pending[ch1]| = 0 < 1)

State 4: CommitRequest(ch1, s1)
  pending = [ch1 |-> {[sender |-> s1, ts |-> 0]}, ch2 |-> {}]
  phase = [<<ch1, s1>> |-> "committed", <<ch1, s2>> |-> "begun"]

State 5: CommitRequest(ch1, s2)
  pending = [ch1 |-> {[sender |-> s1, ts |-> 0],
                      [sender |-> s2, ts |-> 0]}, ch2 |-> {}]
  (VIOLATION: |pending[ch1]| = 2 > MaxPending = 1)

Inv_PendingCap is violated.
```

This trace shows exactly how two concurrent requests can exceed the cap when the check-then-commit is not atomic.