O-Lang Security Model: Conformance vs. Runtime Trust in AI Governance

Author: Olalekan Ogundipe | **Affiliation:** O-Lang Central | **Date:** February 19, 2026

Abstract

> *O-Lang is a deterministic AI workflow language designed to enforce structural safety, auditability, and policy compliance in autonomous systems. A frequent question arises: "If resolvers can be invoked directly during testing, does this undermine O-Lang's security guarantees?" This paper clarifies O-Lang's two-layer trust model—conformance validation and runtime enforcement—and demonstrates how they work together to ensure that only safe, certified capabilities participate in governed AI execution.*

1. Introduction

As AI systems grow more autonomous, ensuring their behavior remains aligned with human intent, policy, and safety constraints becomes critical. O-Lang addresses this by introducing a semantic governance protocol that operates at the execution layer—not merely as a development framework, but as a runtime boundary.

Central to this model are **resolvers**: atomic, side-effect-contained capabilities that perform domain-specific actions (e.g., allocate ICU beds, send Telegram messages, query LLMs). To maintain system integrity, O-Lang strictly separates:

- **Conformance:** Developer-time validation of resolver contracts
- **Trust:** Runtime enforcement by the O-Lang kernel

This separation enables scalable certification without compromising runtime safety.

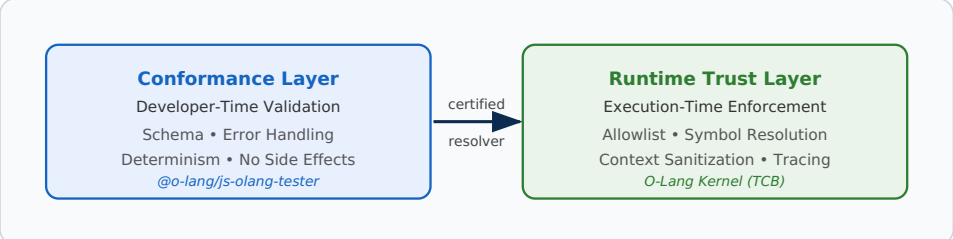

Figure 1 — O-Lang's two-layer security model: conformance validates; the kernel enforces.

2. The Two Layers of O-Lang Security

2.1 Conformance Layer (Developer-Time)

The `@o-lang/js-olang-tester` is the official conformance harness. It validates that a resolver adheres to the **O-Lang Resolver Contract v1.x**, including:

- Input/output schema compliance
- Structured error handling (`{ error: JSON.stringify({ code, message }) }`)
- Deterministic output
- Absence of unmediated side effects

Crucially, this layer invokes the resolver directly as a function. For example:

```
const result = myResolver("Action allocate-icu \"P789\" \"critical\"", context);
```

This is intentional: conformance must be fast, offline, dependency-free, and reproducible. It assumes the developer is acting in good faith and tests only contractual correctness, not adversarial robustness.

> ✅ **Key Point:** Direct invocation during conformance does not bypass security—it validates whether the resolver *deserves* to be trusted.

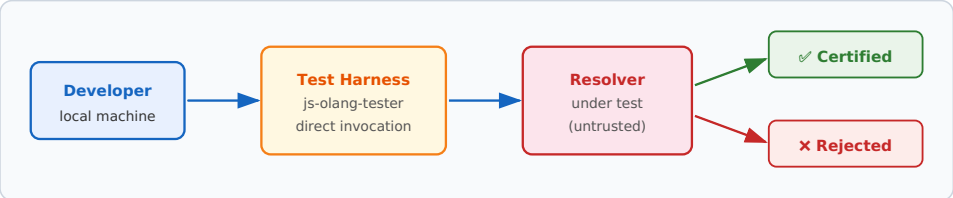

Figure 2 — Conformance testing flow: the harness invokes the resolver directly; results in certification or rejection.

2.2 Runtime Trust Layer (Execution-Time)

In production, no resolver is ever called directly. All invocations flow through the **O-Lang kernel**, which enforces:

- **Allowlist mediation:** Only explicitly declared resolvers may be used
- **Symbol resolution:** Action strings are parsed and validated before dispatch
- **Context sanitization:** Inputs are interpolated safely; no raw template injection
- **Deterministic tracing:** Every step is logged for auditability

For example, a workflow:

```
Workflow "ICU Triage" with patient_id, urgency
Allow resolvers: allocate-icu
Step 1: Action allocate-icu "{patient_id}" "{urgency}"
```

…will only invoke `allocate-icu` if it appears in the allowlist. Even then, the kernel mediates the call—it never exposes raw resolver functions to untrusted inputs.

> 🔒 **Key Point:** The kernel is the sole trusted computing base (TCB). Resolvers are treated as untrusted, external services.

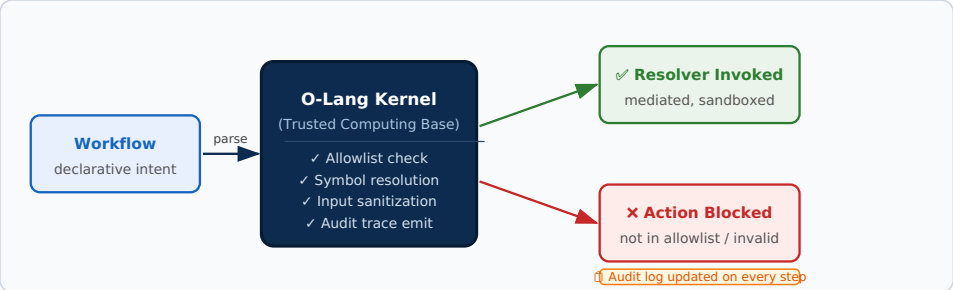

Figure 3 — Runtime enforcement: all execution flows through the kernel; resolvers are never called directly.

---

3. Addressing Common Questions

Q: "Can a resolver be called without the kernel?"

During conformance testing: Yes — but only by the local, trusted developer using the official test harness. This is equivalent to unit testing a library function.

In production or shared environments: No — the kernel always mediates all calls. Unlisted resolvers are rejected at parse time.

Q: "Does direct testing create a security hole?"

No. Conformance testing is not part of the attack surface. It occurs on the developer's machine, before deployment, and without network exposure. Runtime safety is guaranteed by the kernel—not by obscurity or invocation barriers.

Q: "Why not use cryptographic tokens or signatures?"

O-Lang's threat model assumes code integrity (via package signing, supply chain security) rather than per-call authentication. Governance is enforced via allowlists and deterministic mediation, not secrets. This avoids key management complexity and aligns with O-Lang's goal of transparent, auditable execution.

---

4. Formal Guarantees

O-Lang provides the following security properties:

Property	Mechanism
Capability Isolation	Resolvers cannot access global state or other resolvers
Input Sanitization	All variables are interpolated by the kernel before reaching resolvers
Policy Enforcement	Allowlists are checked at workflow parse time
Auditability	Every action produces a deterministic trace
Fail-Safe Defaults	Missing/invalid inputs produce structured errors—never silent failures

These guarantees hold regardless of resolver implementation quality, because the kernel enforces them universally.

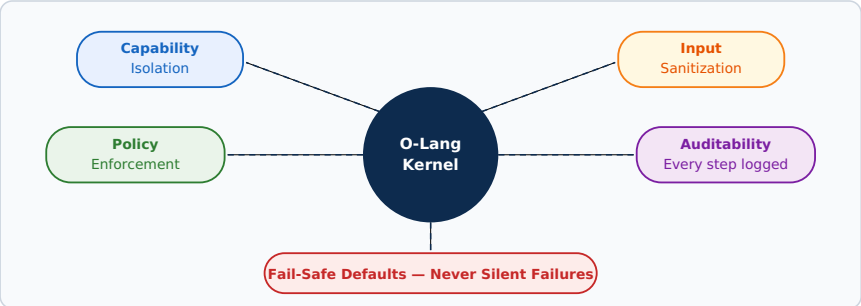

Figure 4 — O-Lang's five formal security properties, all enforced universally by the kernel.

---

5. Conclusion

O-Lang's security model is built on a clear separation of concerns:

- **Conformance** answers: "Does this resolver follow the rules?"
- **Runtime trust** answers: "Is this resolver allowed to run here?"

Direct resolver invocation during testing is not a flaw—it is a necessary enabler of scalable, open certification. In production, the kernel ensures that only conformant, allowlisted resolvers execute—and always under strict mediation.

This architecture makes O-Lang uniquely suited for high-stakes domains like healthcare, finance, and public infrastructure, where structural safety is non-negotiable.

References

1. O-Lang Specification v1.1. https://o-lang-central.github.io/spec/v1.1.html
2. @o-lang/js-olang-tester. https://www.npmjs.com/package/@o-lang/js-olang-tester
3. Ogundipe, O. "Semantic Governance for Autonomous AI." Medium, 2025.

For feedback or collaboration: info@datasciencenigeria.org