

Execution-Witness Binding: Proof-Carrying Authorization for LLM Agent Runtimes

Shawn Cohen

MVAR Security | Universal Media

shawn@mvar.io

February 2026

Abstract

Agentic AI systems that execute privileged actions via tool calls face a fundamental security challenge: autoregressive language models process all tokens uniformly, making deterministic command-data separation unattainable through training alone. Recent concurrent work (Bhattarai & Vu 2026; Kolluri et al. 2026) has independently established that deterministic architectural enforcement is necessary for trustworthy agent deployment, validating the problem space that motivated this work. However, existing proposals evaluate policy at the execution boundary, introducing time-of-check-time-of-use (TOCTOU) vulnerabilities and stateful policy side effects when composition-aware risk tracking is employed. We present MVAR (MIRRA Verified Agent Runtime), an open-source enforcement layer whose architecture was developed independently from these concurrent proposals, originating from the author's prior work on stateful temporal systems (2025). MVAR addresses limitations in per-call evaluation through three novel mechanisms: (1) a **composition risk engine** that tracks cumulative session-level risk across tool chains via sliding-window scoring, detecting stealthy multi-step laundering attacks invisible to prior per-call models; (2) **execution-witness binding**, a proof-carrying code approach that separates planning-phase policy evaluation from execution-phase authorization via cryptographic witness verification, eliminating TOCTOU vulnerabilities and double-evaluation side effects; and (3) **one-time execution token replay defense** with persistent nonce consumption across process restarts. We validate these mechanisms against a reproducible 50-vector adversarial corpus spanning 9 attack categories, with all enforcement behavior CI-regression-protected. MVAR is available under Apache 2.0 at <https://github.com/mvar-security/mvar>.

Keywords: agent runtime security, deterministic enforcement, prompt injection, information flow control, proof-carrying code, execution witness, stateful policy TOCTOU, composition attacks, replay defense

1. Introduction

Modern AI agents operate with increasing authority—executing shell commands, accessing databases, sending emails, and invoking external APIs. When a large language model (LLM) processes both trusted instructions and untrusted data within a unified token stream, adversarial inputs can steer privileged tool execution. This class of vulnerability, broadly termed *prompt injection*, represents not a filtering problem but a control problem: the absence of structural boundaries between influence and authority.

The need for deterministic enforcement at the architectural level—rather than relying on probabilistic model-layer defenses—has been recognized by multiple independent research efforts in early 2026. Bhattarai and Vu (2026) formalize the *Lethal Trifecta*—the co-presence of untrusted inputs, privileged

data access, and external action capability—and propose the Trinity Defense Architecture. Kolluri et al. (2026) demonstrate that IFC-based defenses can provably block unsafe actions but note the cost to task completion rates. The convergence of these independent efforts on the same foundational insight—that deterministic architectural mediation is a necessary condition for trustworthy agent deployment—underscores both the urgency of the problem and the maturity of the solution space.

MVAR's architecture originated independently from these concurrent proposals, growing out of the author's prior work on stateful temporal systems (2025), which established that temporal context tracking with sliding windows produces qualitatively different system behavior than stateless per-event evaluation. The recognition that agent security required deterministic enforcement—rather than content-based filtering—emerged from this foundational work and from the growing body of industry research identifying prompt injection as an unsolved architectural problem.

While MVAR shares the foundational commitment to deterministic enforcement with these concurrent proposals, the specific mechanisms presented here go beyond the shared foundation. Both the Trinity Defense and the IFC-autonomy framework assume that policy evaluation occurs at the execution boundary—a design that introduces two critical problems when combined with stateful policy mechanisms:

- **Time-of-check-time-of-use (TOCTOU) vulnerability:** System state (provenance graphs, composition risk scores, capability grants) can change between policy evaluation and tool execution, invalidating the security decision.
- **Stateful side effects on re-evaluation:** When policy engines maintain session-level state (e.g., cumulative risk scores), re-evaluating the same tool call at the adapter boundary increments counters twice, producing false positives and incorrect risk assessments.

This paper presents MVAR's three novel mechanisms that address these limitations. First, a **composition risk engine** that extends per-call evaluation with session-level cumulative risk tracking, detecting multi-step attack chains invisible to independent call analysis. Second, **execution-witness binding**, which applies proof-carrying code principles (Necula & Lee 1996) to agent authorization: policy evaluation occurs once during planning, producing a cryptographically signed witness that the execution boundary verifies without re-evaluation. Third, **one-time execution token replay defense** with persistent nonce consumption that survives process restarts.

All enforcement behaviors are validated against a reproducible 50-vector adversarial corpus and protected by CI regression gates. MVAR provides first-party adapter surfaces for seven major agent frameworks (LangChain, OpenAI, MCP, Claude, AutoGen, CrewAI, OpenClaw) with conformance harnesses ensuring consistent enforcement across deployment targets.

2. Background and Related Work

2.1 The Agent Security Problem

LLM-based agents extend language models from text generation to autonomous action. The agent receives a user instruction, reasons over it, and produces structured tool calls—function invocations with arguments that are dispatched to external systems. Because the LLM processes all input tokens uniformly (instructions, retrieved documents, user data, and adversarial payloads occupy the same attention space), there is no model-internal mechanism to enforce provenance-aware authorization. Training-based defenses (instruction tuning, RLHF, system prompts) are content-based and therefore forgeable: they reduce empirical attack rates but cannot provide deterministic guarantees.

2.2 Information Flow Control for Agents

Information flow control (IFC) (Denning 1976) provides a principled foundation for tracking data provenance through computational systems. By assigning integrity and confidentiality labels to data and propagating them conservatively through operations, IFC enables policy decisions based on data origin rather than content inspection. Prior IFC systems (Jif [8], FlowCaml [9], Capsicum [7]) demonstrate that label-based enforcement can provide strong security guarantees in traditional computing environments. Applying these principles to agent runtimes requires adapting label semantics to the agent execution model: tool calls carry provenance labels derived from their input lineage, and execution sinks are classified by risk level.

2.3 Capability-Based Security

Capability-based security (Dennis & Van Horn 1966) replaces ambient authority with explicit, unforgeable tokens that grant specific access rights. In agent runtimes, this translates to deny-by-default tool access: each tool-target pair requires an explicit capability grant, and no tool call proceeds without a matching capability. MVAR implements capabilities as per-target enforcement with scope-bound, time-limited execution tokens.

2.4 Concurrent Work on Deterministic Boundaries

In early 2026, multiple independent research efforts converged on the same foundational insight: that deterministic architectural enforcement is necessary for trustworthy agent deployment. This convergence reflects the maturity of the problem rather than derivation between efforts.

Bhattarai and Vu (2026) argue that probabilistic compliance is insufficient for authorization security and propose the Trinity Defense Architecture: action governance via finite action calculus, information flow control via label lattices, and privilege separation between planner and worker components. Their work provides a valuable theoretical framework for reasoning about the necessary conditions for secure agent execution.

Kolluri et al. (2026) build on IFC-based defenses and introduce autonomy metrics to quantify the trade-off between security and task completion, demonstrating that deterministic defenses can be made practical with appropriate planning-phase optimization.

Jiang et al. (2026) systematize runtime supply chain risks for agentic systems, categorizing threats into data and tool supply chain attacks, providing complementary threat modeling. Zhu et al. (2026) propose the OpenPort Protocol for governance-first tool access specifications, addressing authorization and discovery at the API layer.

Both the Trinity Defense and the IFC-autonomy framework assume per-call evaluation at the execution boundary. Neither addresses composition attacks across tool chains, TOCTOU prevention in stateful policies, execution-witness binding, or persistent replay defense. These are the specific gaps that MVAR's novel mechanisms address.

2.5 Proof-Carrying Code

Proof-carrying code (PCC) (Necula & Lee 1996) enables a code producer to attach a machine-checkable proof of safety to mobile code, which a consumer verifies without re-running the analysis. The key insight—separating expensive proof production from cheap proof verification—has not previously been applied to agent tool authorization. We adapt this paradigm: the planning phase produces a signed execution witness (the proof), and the execution boundary verifies the witness without re-evaluating policy (the verification).

3. MVAR Architecture

MVAR interposes a deterministic enforcement layer between the LLM planner and tool execution. The core invariant is: **UNTRUSTED + CRITICAL** → **BLOCK**. Policy outcomes depend on data provenance and sink classification, not payload inspection. The architecture comprises four enforcement mechanisms, three of which (Sections 3.2–3.4) are novel contributions that extend the shared foundational commitment to deterministic enforcement in directions not addressed by concurrent proposals.

3.1 Base Enforcement Layer

The base layer implements provenance-based IFC with integrity and confidentiality labels, capability-constrained tool access with deny-by-default enforcement, and deterministic sink policy evaluation that maps provenance labels and sink risk levels to ALLOW, BLOCK, or STEP_UP outcomes. All policy decisions are signed via QSEAL (Ed25519) for cryptographic auditability. This layer addresses the architectural requirements that multiple independent efforts have converged on as necessary for secure agent execution, and provides the foundation for the novel mechanisms described below.

3.2 Composition Risk Engine

Existing proposals evaluate each tool call independently. This misses a critical attack class: multi-step chains where each individual action is low-risk but the sequence achieves a high-risk outcome. For example, an injected instruction might direct an agent to (1) read a sensitive file, (2) encode its contents, and (3) transmit the encoded data to an external endpoint. Each step in isolation may receive an ALLOW decision; the chain as a whole constitutes data exfiltration.

The composition risk engine originates from the author's prior work on stateful temporal systems, which demonstrated that evaluating cumulative patterns within sliding time windows produces qualitatively different detection capability than stateless per-event evaluation. Applied to agent security, this insight yields a session-aware enforcement mechanism that tracks cumulative risk per principal within a configurable sliding time window.

Each tool call's sink classification contributes a weighted score (LOW=1, MEDIUM=3, HIGH=6, CRITICAL=10). When the cumulative score crosses a step-up threshold, policy outcomes are hardened from ALLOW to STEP_UP (requiring human approval). When the score crosses a block threshold, outcomes are hardened to BLOCK. Events outside the time window are pruned, bounding memory usage to $O(\text{events_in_window})$ per principal and enabling natural cooldown after legitimate high-activity periods.

This mechanism transforms MVAR from a stateless per-call policy engine into a stateful session-aware intrusion detection system for agent runtimes. The sliding window, weighted scoring, and deterministic hardening rules are CI-regression-protected against behavioral drift.

3.3 Execution-Witness Binding

When policy evaluation is stateful (as required by composition risk tracking), evaluating policy at the execution boundary introduces two problems. First, a TOCTOU vulnerability: provenance graphs, capability grants, or risk scores may change between the planning phase (when the LLM decides to invoke a tool) and the execution phase (when the adapter dispatches the call). Second, a double-counting bug: if the adapter re-evaluates policy, the composition risk engine increments the cumulative score twice for the same logical action.

MVAR addresses both problems by adapting the proof-carrying code paradigm to agent authorization. During the planning phase, the policy engine evaluates the tool call and produces a **PolicyDecision** that serves as a cryptographically bound **execution witness**. The witness is a structured object containing the following fields:

```

ExecutionWitness { decision_id: string // unique identifier tool: string // tool name (e.g.,
"bash") action: string // action type (e.g., "exec") target_hash: string // SHA-256 of target
argument provenance_node_id: string // source provenance node policy_hash: string // hash of
active policy config integrity_level: enum // TRUSTED | UNTRUSTED sink_sensitivity: enum // LOW
| MEDIUM | HIGH | CRITICAL composition_score: float // cumulative risk at eval time outcome:
enum // ALLOW | BLOCK | STEP_UP timestamp: ISO-8601 // evaluation timestamp ttl_seconds: int //
witness validity window signature: hex // Ed25519 (QSEAL-compatible) public_key: hex //
verification key }

```

Figure 1. Execution-witness structure. All fields are signed via Ed25519 (QSEAL-compatible).

At the execution boundary, the adapter calls *authorize_execution()* with the pre-evaluated decision as a witness parameter. The policy engine verifies that the witness binds to the current call: tool/action/target must match, provenance node must be unchanged, and policy hash must be current. If any binding check fails, the outcome is deterministically hardened to BLOCK. Critically, the authorization path does **not** re-run the full evaluation pipeline and does **not** increment the composition risk score, guaranteeing single-evaluation semantics.

This design provides three guarantees: (1) **TOCTOU prevention**—any state change between evaluation and execution invalidates the witness; (2) **single-evaluation**—composition risk scores are incremented exactly once per logical action regardless of adapter retry behavior; (3) **fail-closed authorization**—missing or invalid witnesses produce BLOCK outcomes.

Witness verification proceeds as follows:

```

function authorize_with_witness(witness, tool, action, target): // Step 1: Binding check if
witness.tool != tool OR witness.action != action OR SHA256(target) != witness.target_hash:
return BLOCK (reason: witness_binding_mismatch) // Step 2: Signature verification payload =
canonical_serialize(witness.fields) if NOT Ed25519_verify(witness.public_key, payload,
witness.signature): return BLOCK (reason: witness_signature_invalid) // Step 3: Temporal
validity if NOW() - witness.timestamp > witness.ttl_seconds: return BLOCK (reason:
witness_expired) // Step 4: State consistency if current_policy_hash() != witness.policy_hash:
return BLOCK (reason: policy_drift_detected) if
provenance_node(witness.provenance_node_id).changed_since(witness.timestamp): return BLOCK
(reason: provenance_changed) // Step 5: Outcome check if witness.outcome == BLOCK: return BLOCK
(reason: witness_outcome_block) // Authorized: no re-evaluation, no composition score increment
return witness.outcome

```

Figure 2. Witness verification pseudocode. Authorization completes in $O(1)$ without policy re-evaluation.

3.4 One-Time Execution Token Replay Defense

MVAR's STEP_UP mechanism issues scope-bound, time-limited execution tokens when human approval is required. To prevent token reuse attacks, consumed tokens are recorded in a nonce store. We extend this with persistent nonce consumption: the consumed-token ledger is written to a JSONL-backed store that survives process restarts, ensuring that a replayed token is rejected even if the MVAR runtime is restarted between the original use and the replay attempt. The persistent store is pruned by TTL to bound storage growth. When enabled via the MVAR_EXECUTION_TOKEN_NONCE_PERSIST flag, nonce state is fsynced to disk on each consumption, providing crash-consistent replay prevention.

3.5 Deterministic Declassification

Information flow control systems must handle cases where data needs to cross security boundaries legitimately—for example, when a session-scoped value must be written to a user-scoped or organization-scoped memory store. Implicit declassification (silently downgrading labels at boundaries) is a known source of security vulnerabilities in IFC systems. MVAR addresses this with explicit deterministic declassification: every cross-scope data movement requires a signed, one-time-use declassification token specifying the source scope, target scope, authorized principal, and TTL. Sensitive and secret cross-scope writes to memory sinks are blocked unless a valid declassification token is presented and consumed. The token is bound to a single use via nonce tracking, preventing

replay. This ensures that no implicit label downgrade occurs and that every declassification event is auditable.

3.6 Signed Policy Bundle Verification

To prevent policy tampering between deployment and runtime, MVAR supports signed policy bundles. At build or deployment time, the policy configuration is canonicalized and signed via HMAC. At startup, the runtime verifies the bundle signature and compares the policy hash against the expected value. If the bundle is missing, the signature is invalid, or the hash does not match, the runtime refuses to start. This ensures that the policy enforced at runtime is identical to the policy approved at deployment time, closing the gap between policy authoring and policy enforcement. A CLI tool (*generate_signed_policy_bundle.py*) is provided for bundle generation in CI/CD pipelines.

4. Comparison with Concurrent Work

Table 1 summarizes the architectural differences between MVAR and two closely related concurrent proposals. All three systems share a foundational commitment to deterministic enforcement—a convergence that reflects independent recognition of the same underlying problem rather than derivation between efforts. The comparison highlights the specific mechanisms where MVAR extends this shared foundation.

Property	Trinity Defense (Bhattarai & Vu)	IFC + Autonomy (Kolluri et al.)	MVAR (This Work)
Evaluation timing	At execution	At execution	At planning
Authorization method	Policy re-check	Policy re-check	Witness verification
TOCTOU protection	No	No	Yes (crypto binding)
Composition detection	No (per-call)	No (per-call)	Yes (session-level)
Stateful side effects	N/A (stateless)	N/A (stateless)	Eliminated (single-eval)
Proof-carrying auth	No	No	Yes (witness = proof)
Replay prevention	Not addressed	Not addressed	Persistent nonce store
Declassification	Not addressed	Not addressed	Explicit token-gated
Policy integrity	Not addressed	Not addressed	Signed bundle gate
Framework adapters	None (theory)	1 (prototype)	7 (conformance-tested)
Adversarial corpus	None	AgentDojo / WASP	50 vectors, 9 categories
CI regression gates	No	No	Yes (all behaviors)
Open source	No	No	Yes (Apache 2.0)
Cryptographic audit	Not specified	Not specified	QSEAL Ed25519 signing

Table 1. Architectural comparison of concurrent deterministic enforcement approaches for LLM agent runtimes. All three systems share the foundational commitment to deterministic enforcement. MVAR extends this shared foundation with stateful composition detection, proof-carrying witness authorization, deterministic declassification, signed policy bundles, and persistent replay defense.

5. Validation

5.1 Adversarial Corpus

MVAR is validated against a reproducible 50-vector adversarial corpus spanning 9 attack categories: direct prompt injection, indirect prompt injection, credential exfiltration, encoded/obfuscated payloads,

tool override attempts, multi-stage indirect attacks, privilege escalation, data poisoning, and composition chain attacks. The corpus is designed to characterize deterministic enforcement behavior under controlled conditions; it is not a completeness claim against all possible attacks. The full corpus and validation harness are included in the repository for independent reproduction via `./scripts/launch-gate.sh`.

5.2 Composition Risk Validation

We validate the composition risk engine against multi-step attack chains that evade per-call evaluation. Test scenarios include: (1) sequential low-risk calls that individually pass but cumulatively trigger STEP_UP or BLOCK thresholds; (2) session risk budget exhaustion across diverse tool types; (3) sliding window pruning that correctly resets scores after the configured time window expires. All composition risk behaviors are CI-regression-protected: the test suite runs on every commit and PR, preventing behavioral drift.

5.3 Witness Binding Validation

Execution-witness binding is validated through conformance tests that verify: (1) tool/action/target mismatch between witness and execution call produces BLOCK; (2) the pre-evaluated authorization path does not increment composition risk scores (single-eval guarantee); (3) missing witnesses fall through to full evaluation (fail-safe behavior). These tests are integrated into the adapter conformance harness, ensuring consistent witness behavior across all seven framework adapters.

5.4 Replay Defense Validation

One-time token replay defense is validated through a red-team regression gate that simulates process restart between token issuance and replay attempt. The test creates a policy instance, issues and consumes a token, then instantiates a new policy instance (simulating restart) and verifies that the persistent nonce store correctly rejects the consumed token. This closes the replay window that exists in memory-only nonce tracking implementations.

5.5 Declassification Validation

Deterministic declassification is validated through regression tests covering: (1) cross-scope writes without a declassification token are blocked; (2) valid declassification tokens permit the specified scope transition; (3) consumed declassification tokens cannot be replayed. Scope widening enforcement is tested across session, user, and organization boundaries for memory write sinks at sensitive and secret classification levels.

5.6 Policy Bundle Validation

Signed policy bundle verification is validated through tests covering: (1) runtime startup with a valid signed bundle succeeds; (2) startup with a missing bundle is blocked; (3) startup with a tampered or mismatched bundle is blocked. These tests ensure that policy integrity is enforced before any tool call is evaluated.

5.7 Agent Testbed Trilogy

A three-scenario agent testbed provides end-to-end validation of the enforcement pipeline. The trilogy compares baseline (no enforcement) against MVAR-enforced execution for: (1) *injection*—adversarial command in untrusted input, baseline ALLOWs, MVAR BLOCKs; (2) *taint laundering*—data derived from untrusted source attempts privileged action, baseline ALLOWs, MVAR BLOCKs; (3) *benign*—legitimate low-risk operation, both baseline and MVAR ALLOW. This trilogy is CI-gated on every commit, ensuring the fundamental enforcement invariant is never violated.

6. Threat Model and Limitations

MVAR's threat model and known limitations are documented in `THREAT_MODEL.md` in the repository. We summarize key assumptions and scope boundaries here to enable informed deployment decisions and to guide community contributions:

- **Sink registration is manual.** MVAR requires explicit annotation of execution sinks. Unregistered sinks bypass enforcement. Automatic sink discovery via tool invocation tracing with CI coverage enforcement is planned future work.
- **Composition risk scoring uses fixed weights.** The current weight assignment (LOW=1, MEDIUM=3, HIGH=6, CRITICAL=10) is configurable but not learned. Adaptive weight tuning based on deployment-specific attack distributions is future work.
- **The adversarial corpus is not exhaustive.** The 50-vector corpus characterizes deterministic behavior under known attack patterns. Novel attack vectors outside the corpus may not be represented. Community vector submission is supported via `ATTACK_VECTOR_SUBMISSIONS.md`.
- **Witness binding assumes honest enforcement layer.** If the enforcement layer itself is compromised (e.g., an attacker modifies the MVAR runtime), witness binding does not provide protection. MVAR assumes the enforcement layer is trusted; the LLM planner is untrusted. This is consistent with the standard reference monitor assumption (Anderson 1972).
- **Composition attacks with sub-threshold chains.** An attacker who can keep each individual step and the cumulative score below the configured thresholds will evade composition detection. Threshold tuning guidance and deployment-specific risk profiling are areas for future investigation.

These limitations reflect Phase 1 scope decisions. Documenting them explicitly follows the principle that honest threat modeling strengthens rather than weakens a security system's credibility.

7. Design Lineage and Independent Development

MVAR's architecture was developed independently of the concurrent proposals cited in this paper. The development history, preserved in the project's version control and documented in `DESIGN_LINEAGE.md`, establishes the following timeline and lineage:

The foundational architectural insights originated from the author's prior work on stateful temporal systems (2025), which established that sliding-window context tracking across sessions produces qualitatively different system behavior than stateless per-event evaluation. The application of this architectural insight to agent security—tracking cumulative risk patterns rather than evaluating independent snapshots—directly informed the composition risk engine (sliding-window cumulative scoring), QSEAL signing (cryptographic decision auditability), and provenance tracking (data lineage through transformations).

The recognition that agent security required deterministic enforcement at the architectural level—rather than content-based filtering at the model layer—emerged from this foundational work and from the growing body of industry research identifying prompt injection as an unsolved structural problem in agentic systems. The specific insight that untrusted data must be tracked by provenance rather than inspected by content led naturally to IFC-based enforcement, drawing on the classical systems security literature (Denning 1976, Dennis & Van Horn 1966, Anderson 1972).

Bhattarai and Vu (2026) and Kolluri et al. (2026), published in February 2026, arrived independently at the same foundational conclusion: that deterministic architectural boundaries are necessary for trustworthy agent execution. This convergence of multiple independent efforts on the same insight reflects the maturity and urgency of the problem rather than derivation between efforts. We cite these

concurrent proposals as related work that validates the shared foundational commitment to deterministic enforcement, while noting that MVAR's specific mechanisms—composition risk detection, execution-witness binding, deterministic declassification, signed policy bundles, and persistent replay defense—are novel contributions not present in either concurrent proposal.

The second line of intellectual ancestry draws from the proof-carrying code paradigm (Necula & Lee 1996). The application of PCC principles to agent authorization—separating planning-phase policy evaluation (proof production) from execution-phase witness verification (proof checking)—is, to our knowledge, a novel contribution of this work.

8. Conclusion

We have presented MVAR, an open-source enforcement layer for LLM agent runtimes that extends the deterministic architectural boundary paradigm—independently arrived at by multiple concurrent research efforts—with novel mechanisms that address concrete limitations in per-call, stateless policy evaluation.

MVAR's contributions include: composition risk detection via session-level cumulative scoring with sliding-window temporal tracking; execution-witness binding adapted from the proof-carrying code paradigm for TOCTOU-proof authorization with single-evaluation guarantees; persistent one-time token replay defense that survives process restarts; deterministic declassification via explicit, token-gated scope transitions; and signed policy bundle verification ensuring deployment-to-runtime policy integrity.

These mechanisms address gaps in concurrent proposals that assume stateless, per-call policy evaluation at the execution boundary. The composition risk engine detects multi-step attack chains that evade independent call analysis. Execution-witness binding eliminates TOCTOU vulnerabilities and stateful side effects. Deterministic declassification prevents implicit label downgrade. Signed policy bundles close the policy tampering gap between deployment and runtime.

All enforcement behaviors are validated against a reproducible adversarial corpus and protected by CI regression gates. MVAR provides adapter surfaces for seven major agent frameworks with conformance harnesses ensuring cross-framework consistency. The system is available under Apache 2.0 at <https://github.com/mvar-security/mvar>.

Future work includes automatic sink discovery with CI coverage enforcement, adaptive composition risk weight tuning, transparency log mode with Merkle-rooted decision digests, formal verification of the witness binding protocol, distributed nonce ledger support for multi-instance deployments, and composition attack modeling across agent delegation chains.

References

- [1] Anderson, J. P. (1972). Computer security technology planning study. Technical Report ESD-TR-73-51, Air Force Electronic Systems Division.
- [2] Bhattarai, M. & Vu, M. (2026). Trustworthy agentic AI requires deterministic architectural boundaries. [arXiv:2602.09947](https://arxiv.org/abs/2602.09947).
- [3] Denning, D. E. (1976). A lattice model of secure information flow. *Communications of the ACM*, 19(5), 236–243.
- [4] Dennis, J. B. & Van Horn, E. C. (1966). Programming semantics for multiprogrammed computations. *Communications of the ACM*, 9(3), 143–155.
- [5] Kolluri, A. et al. (2026). Optimizing agent planning for security and autonomy. [arXiv:2602.11416](https://arxiv.org/abs/2602.11416).
- [6] Necula, G. C. & Lee, P. (1996). Safe kernel extensions without run-time checking. *Proceedings of the 2nd OSDI*, 229–243.
- [7] Watson, R. N. M. et al. (2010). Capsicum: Practical capabilities for UNIX. *Proceedings of the 19th USENIX Security Symposium*.

- [8] Myers, A. C. (1999). JFlow: Practical mostly-static information flow control. Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.
- [9] Simonet, V. (2003). The Flow Caml system: Documentation and user's manual. Technical Report 0282, INRIA.
- [10] Jiang, X. et al. (2026). Agentic AI as a cybersecurity attack surface: Threats, exploits, and defenses in runtime supply chains. [arXiv:2602.19555](#).
- [11] Zhu, G. et al. (2026). OpenPort Protocol: A security governance specification for AI agent tool access. [arXiv:2602.20196](#).

Appendix A: Reproduction Instructions

All results reported in this paper can be reproduced locally:

```
git clone https://github.com/mvar-security/mvar.git cd mvar python3 -m venv .venv && source
.venv/bin/activate pip install -e ".[dev]" ./scripts/launch-gate.sh
```

The launch gate script executes the full validation suite including red-team gate tests, the 50-vector adversarial corpus, composition risk regression tests, witness binding conformance tests, declassification tests, policy bundle verification, agent testbed trilogy, and replay defense verification. Expected runtime is approximately 2 minutes on commodity hardware. Individual test suites can be run independently via pytest.

Appendix B: Configuration Reference

```
# Composition risk engine export MVAR_ENABLE_COMPOSITION_RISK=1 export
MVAR_COMPOSITION_STEP_UP_THRESHOLD=8 # ALLOW → STEP_UP export
MVAR_COMPOSITION_BLOCK_THRESHOLD=12 # → BLOCK export MVAR_COMPOSITION_WINDOW_SECONDS=900 #
15-minute sliding window # Persistent nonce store export MVAR_EXECUTION_TOKEN_NONCE_PERSIST=1 #
Enable persistent replay defense # Signed policy bundle verification export
MVAR_REQUIRE_SIGNED_POLICY_BUNDLE=1 # Require valid bundle at startup export
MVAR_POLICY_BUNDLE_PATH=policy_bundle.json export MVAR_POLICY_BUNDLE_SECRET=<your-hmac-secret>
```

Risk weights are assigned by sink classification: LOW=1, MEDIUM=3, HIGH=6, CRITICAL=10. Thresholds and window duration are configurable per deployment. Events outside the sliding window are pruned from the cumulative score, enabling natural cooldown after legitimate high-activity periods. The persistent nonce store writes consumed tokens to a JSONL file that is loaded on startup and pruned by TTL. Policy bundles are generated via the included CLI tool and verified at runtime startup before any tool call is evaluated.

Appendix C: Sample Execution Witness

The following is a representative execution witness as produced by MVAR's policy engine during planning-phase evaluation:

```
{ "decision_id": "evt_a3f8c912", "tool": "bash", "action": "exec", "target_hash":
"e3b0c44298fclcl49afbf4c8996fb924...", "provenance_node_id": "prov_node_7b2a", "policy_hash":
"647d27da00eflca9a58470c287ce2e3a...", "integrity_level": "UNTRUSTED", "sink_sensitivity":
"CRITICAL", "composition_score": 4.0, "outcome": "BLOCK", "reason": "UNTRUSTED integrity +
CRITICAL risk sink = BLOCK", "timestamp": "2026-02-27T14:30:00Z", "ttl_seconds": 300,
"signature": "alb2c3d4e5f6... (Ed25519 / QSEAL-compatible)", "public_key": "04f7a8b9c0d1...
(Ed25519 public key)" }
```

Figure 3. Sample execution witness (JSON). The signature covers all fields except signature and public_key. Verification requires only the public key and the witness payload, completing in $O(1)$ without policy re-evaluation.

When this witness is presented at the execution boundary, the adapter calls *authorize_execution()* which verifies the signature, checks tool/action/target binding, confirms temporal validity, and returns the witnessed outcome without incrementing composition risk state. A mismatch on any field produces a deterministic BLOCK with reason *pre_evaluated_decision_mismatch*.