LARA: VERIFIER-GUIDED RULE-LEVEL SEMANTIC FUZZING FOR AUDITING LOCAL AGENT RUNTIMES

Anonymous authors
Paper under double-blind review

ABSTRACT

Local LLM agents mediate between partially trusted inputs and privileged host actions through runtime code. Historical fixes reveal failures in this mediation, but their code-level forms are project-specific. We present LARA (Local Agent Runtime Auditor), which formulates security-rule synthesis as verifier-guided, rule-level semantic fuzzing: semantic security rules are the evolving artifacts rather than program inputs; applicability and validity reasoning drive residual-conditioned CULL, MODIFY, and AUGMENT mutations rather than fixed operators; and vulnerable/fixed execution feeds new rounds until semantic saturation or the round budget is exhausted. Accepted rules are grounded in a runtime and executed as CodeQL queries. On 275 OpenClaw construction cases, this search raises Patch Recall from 61.1% to 81.5% and patch-discriminating Success from 61.1% to 80.0%; an ablation attributes 49 of the 52 additional successful cases to semantic rule revision rather than query-only refinement. Before target vulnerability feedback, adapted rules localize 17/22 Nanobot and 9/12 Hermes repairs, but discriminate only 5/22 and 4/12, respectively. Refinement on each target's historical cases raises Success to 15/22 and 7/12. Separately, review of frozen-corpus findings on four latest snapshots produced 25 previously unreported vulnerability candidates validated by proof-of-concept (PoC) exploits. These results suggest that reusable security rules from historical fixes can bootstrap the auditing of new local agent runtimes, for which target-agent-specific evidence and rule adaptation is essential.

1 INTRODUCTION

Local LLM agents such as OpenClaw Steinberger & OpenClaw Contributors (2026) and Nanobot HKUDS (2026) are installable programs with direct access to the shell, filesystem, browser, credentials, and messaging applications. Their runtime code accepts partially trusted input, constructs prompts, parses model output, dispatches tools, loads skills, writes memory, and opens network connections. Security therefore depends not only on model behavior, but on whether this privileged runtime preserves the trust boundaries from input to effect.

Existing work on agent safety examines prompt injection and model-facing attacks Greshake et al. (2023); Zhan et al. (2025); Chang et al. (2025), malicious skills and marketplace risks Liu et al. (2026b); Beurer-Kellner et al. (2026a); Koi Security (2026), and agent behavior under attack Arora et al. (2025); Patir et al. (2026). Recent studies examine agent frameworks and MCP servers Zhang et al. (2026); Ma et al. (2026); Hasan et al. (2025); Chen et al. (2026). The runtime implementation raises a complementary problem: expressing the security requirements of its mediating components as rules that remain useful beyond one flaw or one agent.

Historical vulnerability/fix pairs expose a failed requirement and its repair, allowing an executable query to be checked on both revisions. Yet a query that copies patch-local syntax, helper names, or APIs can recognize one repair without expressing the requirement it enforces. A reusable rule must instead retain the security-sensitive entity, protected effect, and required relation, then bind those roles to each agent runtime's architecture.

We present LARA (Local Agent Runtime Auditor), which generalizes fuzzing along three axes: a semantic-rule corpus rather than grammar-generated inputs; residual-guided CULL/MODIFY/AUGMENT mutations rather than fixed syntactic operators; and executable resid-

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

ual feedback with saturation-or-budget stopping rather than coverage. Accepted rules execute as CodeQL GitHub (2025) queries. We call this formulation *rule-level semantic fuzzing*.

The synthesizer builds a source corpus from historical fixes and existing rules, then adapts it to independently implemented targets. Each target R_0 is frozen before target vulnerability/fix feedback, separating initial localization from later target-specific refinement. Frozen corpora then run on latest snapshots; contextual review and PoC verification form a separate downstream evaluation.

On 275 OpenClaw *construction* cases, Success rises from 61.1% to 80.0%; because these cases drive the search, this measures construction fit rather than held-out generalization. Before target feedback, Nanobot and Hermes R_0 localize 77.3% and 75.0% of their repair regions but discriminate only 22.7% and 33.3%, respectively. Refinement on the same target histories raises Success to 68.2% and 58.3%. On latest snapshots of OpenClaw, Nanobot, Hermes, and OpenHands, downstream review yields 25 previously unreported vulnerabilities validated by proof-of-concept (PoC) exploits, including 11 on OpenHands using the Nanobot corpus unchanged.

Our contributions are:

- A *semantic rule* representation that separates a reusable security requirement from target bindings and executable queries, organized by a STRIDE-derived runtime taxonomy (Sections 3–4).
- Rule-level semantic fuzzing that searches static analysis rules rather than concrete program test inputs, derives CULL/MODIFY/AUGMENT mutations from applicability and validity reasoning rather than fixed mutation operators as in common fuzzing, and iterates on executable residual feedback until semantic saturation or budget exhaustion; deterministic verification gates accepted rule changes (Section 4).
- Extensive empirical studies that distinguish construction fit, pre-feedback target localization, target-specific refinement, and latest-snapshot auditing. Semantic mutation accounts for 49 of 52 construction gains, while downstream review produces 25 PoC-validated vulnerabilities across four local agent runtimes (Sections 5–6).

2 Related Work and Positioning

Agent-security research emphasizes model-facing attacks and defenses, behavioral benchmarks, malicious skills, agent frameworks, and MCP ecosystems (Greshake et al., 2023; Zhan et al., 2025; 2024; Debenedetti et al., 2024; Liu et al., 2026b; Beurer-Kellner et al., 2026a; Zhang et al., 2026; Hasan et al., 2025; Chen et al., 2026). LARA addresses a complementary unit of analysis: the runtime code mediating between partially trusted inputs and privileged host effects, and the conversion of historical repairs into executable requirements that can be re-grounded in another runtime.

Semgrep encodes syntactic patterns, while CodeQL supports structural and interprocedural data-flow queries (Nong et al., 2021; 2022; Avgustinov et al., 2016). IRIS uses an LLM to infer taint specifications and contextualize CodeQL warnings (Li et al., 2025); QLCoder synthesizes a query for an individual vulnerability from its vulnerable/fixed pair (Wang et al., 2026). LARA instead searches an intermediate corpus of semantic requirements shared across cases: a rule states an entity–effect relation independently of target APIs, may admit several queries, and can be re-grounded without changing the requirement. The contribution is a verifier-guided procedure for revising and transferring this corpus, not another query generator. Because QLCoder and LARA use different synthesis units and evidence regimes, their comparison is contextual rather than a controlled claim of superiority.

3 Agent Runtime Security Model

This model is LARA's foundation: trust boundaries mark where privileged effects need mediation, the taxonomy organizes failure classes, and semantic rules state reusable requirements.

3.1 Runtime Trust Boundaries

An agent runtime mediates between partially trusted input and operations available to its host process. User goals, retrieved content, tool results, and model output can pass through prompt construction, parsing, tool dispatch, skill loading, and memory before reaching a shell, filesystem, browser, or network client. Authentication and permission gates must constrain which caller or execution context

may trigger those effects. The security question concerns the code that connects an input to an action; the model, skills, and sandbox can be analyzed separately.

We examine three transitions: input entering the agent core, the core handing work to tools or skills, and the runtime communicating with external resources. Three questions guide the analysis of each transition: *What attacker- or model-controlled data crosses the boundary? What privileged operation follows? Which component mediates the transition?* The answers identify a security-sensitive effect and the mediation that should constrain it.

3.2 STRIDE-Guided Vulnerability Taxonomy

Enumerating runtime weakness categories by intuition risks both omission and overlap. We therefore derive them from STRIDE Microsoft (2009); Shostack (2014), which applies threat classes to trust boundaries in a data-flow model. The three questions above make the derivation concrete: the first locates the untrusted input, the second locates the privileged consequence, and the third names the runtime component that should mediate the transition.

Applying STRIDE to these boundaries yields four relevant threat classes: Tampering, Elevation of Privilege, Information Disclosure, and Spoofing. Tampering splits into two categories because controlled data reaches the runtime at two structurally different points: untrusted text enters prompt construction, whereas model- or user-controlled operands steer tool and skill execution. The mediating components and the required checks differ; guarding one boundary does not guard the other. As shown in Table 1, the four threat classes yield five runtime categories.

Table 1: STRIDE-guided categories of agent-runtime weaknesses; examples indicate implementation surfaces, not individual rules

STRIDE threat	**Runtime category**	**Representative failure**
Tampering	CAT-1 Prompt Handling	Memory contamination
Tampering	CAT-2 Tool/Skill Execution	Path traversal; command injection
Elevation of Privilege	CAT-3 Permission & Isolation	Missing confinement
Information Disclosure	CAT-4 Network & Communication	SSRF; credential leakage
Spoofing	CAT-5 Authentication & Authorization	Missing caller admission

CAT-1 Prompt Handling. Untrusted text enters prompt construction or memory. An agent assembles its prompt from user goals, retrieved documents, tool outputs, and stored memory, any of which can carry instructions the developer did not intend the model to follow. The mediating components are the prompt builder and memory writer. A weakness here lets external text acquire instruction authority, either in the current turn or later through contaminated memory.

CAT-2 Tool/Skill Execution. Model- or user-controlled data influences the filesystem, shell, or skill loading. The runtime turns an action request into concrete operations by resolving a path, building a command, or loading a skill. The tool dispatcher and skill loader mediate that transition. A crafted argument or skill can escape the intended operation through path traversal, command injection, or unsafe loading. Unlike CAT-1, the missing guard lies between an action request and its host effect.

CAT-3 Permission & Isolation. The runtime must confine tools, skills, and execution contexts even when an individual operation is well formed. Permission gates and isolation mechanisms should scope filesystem access, sandbox a skill, and separate execution contexts. A missing or overly broad check lets an otherwise valid action reach further into the host than it should.

CAT-4 Network & Communication. The runtime constructs requests, logs, and other external interactions. An attacker-influenced URL can redirect an outbound request, while a log line or error message can disclose internal data. Network clients and logging components mediate these emissions. Missing checks can permit SSRF or credential leakage to an unintended destination.

CAT-5 Authentication & Authorization. A runtime that exposes handlers or manages sessions must check who is asking before acting and bind a resource to the session entitled to it. Unlike CAT-3, which confines an admitted action, the admission gate mediates caller identity at entry. A missing or bypassable check lets an unauthenticated or under-privileged caller obtain access.

Repudiation and workload-dependent denial of service are outside our scope. Repudiation concerns audit-log integrity more than boundary mediation, and workload-dependent denial of service is only partially observable to static rules; statically visible resource-control weaknesses, such as a missing size bound, remain in scope. The five categories organize historical evidence and are not exhaustive; the units the rule synthesizer evolves are the semantic rules defined next.

3.3 From Vulnerability Categories to Semantic Rules

A category identifies a class of failure, but not the reusable property code must enforce. CAT-5, for example, does not say which principal must be admitted, which effect is protected, what admission relation must hold, or where that requirement applies. Two runtimes may represent an actor through different sender or session structures. Both may require the same condition: a privileged messaging effect occurs only when the acting principal satisfies the required admission or attestation relation. The concrete fields and APIs differ, but the security requirement can remain stable.

The security model separates the failure class from the reusable requirement: STRIDE/CAT names the class, while a *semantic rule* states the requirement as a relation between a security-sensitive entity and a protected effect.

A semantic rule states a property independently of target function names. Applicability and scope specify where it applies; abstract roles identify the security-sensitive *entity* and *protected effect*; and a required relation specifies their safe interaction. A rule separates these abstract roles from their *target bindings*: the program entities, symbols, or predicates that play each role in a specific runtime. *Grounding* fills these bindings from target-source evidence and, where necessary, sets target scope and applicability, so the security requirement can persist while its grounding changes across runtimes.

A *query* separately implements a rule as code features an analyzer can check. One rule can have several queries when different program shapes express the same property. This separation prevents runtime-specific APIs and query syntax from defining the portable requirement: the rule states what must hold, and its queries look for code where it does not. Section 4 describes how LARA's rule synthesizer constructs and adapts rule corpora and how its audit engine executes their queries.

4 LARA

4.1 Overview

Figure 1 shows LARA's three components: the security model of Section 3, the rule synthesizer, and the audit engine that realizes rules as CodeQL queries and executes them.

LARA closes a loop over corpus states: execute S_n, reason over residuals, propose semantic Cull/Modify/Augment mutations, deterministically admit non-regressing changes to S_{n+1}, and repeat until saturation or budget. This is rule-level semantic fuzzing because the evolving artifact is reusable security knowledge, not a program input. Phase I builds the source corpus; Phase II re-grounds it in each target, freezes R_0 before target vulnerability feedback, and refines it.

4.2 Phase I: Source-Runtime Construction

Historical fixes expose failed security requirements, but their code-level forms are too specific to serve as reusable rules. Phase I synthesizes a seed corpus S_0 from source-runtime vulnerability/fix evidence and existing security rules. The taxonomy organizes this evidence: STRIDE/CAT labels guide deterministic grouping of historical cases and are supplied to seed synthesis as contextual metadata, while each rule's entity, protected effect, and required relation are inferred from the vulnerability evidence itself. The seeds are initial rules for fuzzing; the historical pairs show what they miss.

Executing S_n through the audit engine on vulnerable and repaired revisions yields per-case execution evidence. A *residual* is a historical case that no rule in S_n reconstructs under the verifier's deterministic reconstruction check (Appendix B); the residuals of a round form its *residual frontier*. For each residual, an LLM-guided semantic layer analyzes the entity, effect, required relation, and vulnerable/fixed difference. It excludes patch-specific constructs and, again using taxonomy labels, groups cases that may share a reusable revision. This semantic diagnosis is needed because a fixed syntactic

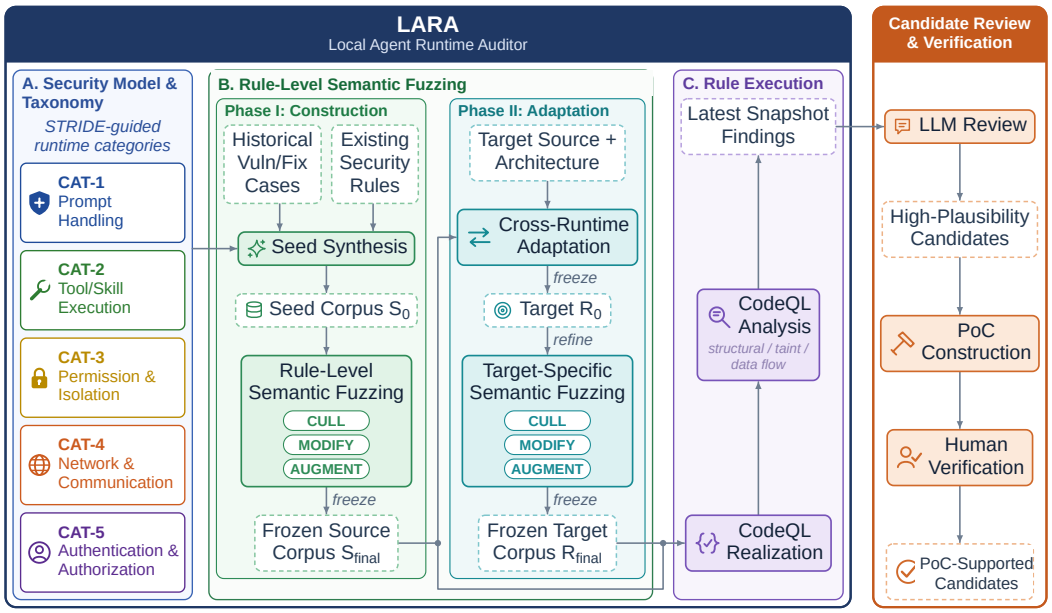

Figure 1: LARA's loop: executable residuals drive semantic rule mutations until saturation or budget; frozen rules drive auditing and verification

mutation operator cannot determine which entity, relation, scope, or applicability assumption the residual calls into question.

Residual reasoning assesses rule applicability and validity, then proposes one of three *semantic action classes*. CULL removes an inapplicable or unsupported rule. MODIFY revises a rule or its grounding, including its entity, effect, required relation, applicability, or scope. AUGMENT introduces a missing reusable rule. These are semantic decisions, and queries are regenerated from each revised rule. The same classes drive target refinement (Section 4.3); on Nanobot, for example, a sender-admission query selected an admission call that survived the fix and fired on both revisions. A MODIFY re-grounded the rule around the unattested identity claim, and one revised rule recovered two independently implemented cases (Appendix B). A fixed syntactic operator would not state that change.

The LLM layer proposes semantic rule revisions but cannot establish that they distinguish vulnerable from repaired code. The audit engine deterministically lowers each grounded rule to CodeQL queries, then compiles and executes them on the historical revisions; seeds imported as CodeQL retain their original queries. An inexpressible proposal is recorded as backend-limited instead of being broadened until it compiles. Compilation, execution on both revisions, regression checking, and acceptance are deterministic, so no LLM proposal enters the corpus on its own authority. A proposal is accepted only if its query compiles, reconstructs and discriminates at least one motivating residual case, and loses no existing Success or Patch Recall. Reconstruction is a separate deterministic predicate over vulnerable-revision findings and the repair region; it is stricter than Patch Recall and does not make the evaluation metrics determine the residual frontier. Accepted changes form S_{n+1}, whose unresolved cases become feedback for the next round.

The intended fixed point is *semantic saturation*: no remaining residual yields a reusable hypothesis under the current representation and backend. Residuals may instead be exhausted, backend-limited, or specialization-only. A run stops at this representation-relative fixed point or at a predefined round budget; neither outcome certifies completeness. The accepted source corpus is frozen as S_{final}.

4.3 PHASE II: CROSS-RUNTIME ADAPTATION

A source rule may retain its security meaning while target components and APIs change. Phase II re-grounds the frozen source corpus in each target rather than translating source API names. It freezes

target R_0 before target vulnerability/fix evidence is supplied. Adaptation may inspect target source, architecture, and runtime maps, so R_0 is *feedback-free*, not zero-shot with respect to target code.

Each source rule receives a KEEP, MODIFY, or CULL decision according to its target applicability. For example, CULL drops a rule whose security surface the target lacks. These decisions describe how a source rule maps to a target; they are distinct from the CULL/MODIFY/AUGMENT revision actions used in semantic fuzzing. Adaptation is performed by the LLM-guided grounding layer using only target source, architecture, and runtime-map evidence. Before R_0 is frozen, grounded rules undergo deterministic realizability and validity checks without target vulnerability/fix feedback; those that pass form R_0. Their localization and vulnerable/fixed discrimination on target historical pairs are measured only after the freeze.

After the R_0 freeze, target vulnerability/fix evidence drives the same residual-guided revision and no-regression checks as in Phase I. Accepted target rounds form a frozen target S_{final}.

4.4 RULE EXECUTION

The audit engine realizes grounded semantic rules as CodeQL queries by deterministic lowering, using structural or taint/data-flow reasoning as appropriate. During synthesis, it executes the queries of proposed revisions on vulnerable and fixed revisions and returns evidence for deterministic validation and residual reasoning. Once a corpus is frozen, the same engine executes its queries on a runtime revision to emit security findings. Section 6 evaluates those findings separately.

5 EVALUATION

We evaluate LARA's rule synthesizer through three research questions:

- **RQ1 — Source-Runtime Construction.** How effectively does LARA construct a reusable security-rule corpus from historical vulnerability evidence?
- **RQ2 — Cross-Runtime Adaptation.** How well does a source-runtime corpus transfer to independently implemented runtimes without target vulnerability feedback?
- **RQ3 — Target-Specific Refinement.** How effectively does target-specific semantic fuzzing refine an adapted corpus after target vulnerability feedback becomes available?

5.1 EXPERIMENTAL SETUP

Subjects and evidence regimes. Each case pairs a vulnerable and fixed revision. OpenClaw (JavaScript/TypeScript) supplies 275 cases for source-corpus construction; Nanobot and Hermes Agent (Python) supply 22 and 12 cases for target-specific refinement. OpenClaw and final target scores quantify fit to historical evidence used by construction or refinement, not held-out generalization. Only target R_0 is evaluated before target vulnerability/fix feedback; latest-snapshot auditing is evaluated separately through review and PoCs.

Metrics. *Patch Recall* is the fraction of cases for which some valid query finds the vulnerable repair region (fix hunks padded by three lines). *Success* additionally requires that same query to stop matching the corresponding fixed region; Appendix B gives the formal definitions. These repair-local metrics are recomputed independently of the residual frontier. They neither penalize findings elsewhere nor estimate false-positive rate, whole-program precision, or exploitability.

Freeze protocol. Freezes follow Section 4: source S_{final} before adaptation, each target R_0 before target historical feedback, and each target final corpus before the latest-snapshot study in Section 6.

Baselines and ablations. Section 5.5 compares LARA with Stock Semgrep, Stock CodeQL, and QLCoder (Wang et al., 2026), which synthesizes one query per vulnerability from its own vulnerable/fixed pair, and with two ablations: *w/o fuzzing* (the seed corpus alone) and *w/o semantic mutation* (rules frozen, query refinement kept).

LARA's LLM layer uses Claude Opus 5 (`claude-opus-5`) with adaptive thinking. More details on the configuration and prompt templates are in Appendix D. The latest-snapshot study pins Nanobot,

Hermes, and OpenHands at their default-branch heads of September 22, 2026, and OpenClaw at its head of September 16, 2026.

5.2 RQ1: SOURCE-RUNTIME CONSTRUCTION

Table 2 reports construction-set fit on the 275 pairs that supply residual feedback. The corpus grows from 166 rules and queries at R_0 to 180 rules implemented by 218 queries at R_8; Patch Recall rises from 61.1% to 81.5% and Success from 61.1% to 80.0%. Most gains occur by R_3 (78.5% Patch Recall; 76.4% Success).

Table 2: OpenClaw construction (275 cases)

Round	Queries	Patch Recall	Success	Findings
R_0	166	61.1%	61.1%	1,001
R_1	189	70.9%	69.5%	2,548
R_2	199	75.6%	73.1%	2,878
R_3	210	78.5%	76.4%	3,151
R_4	211	78.9%	76.7%	3,152
R_5	213	79.3%	77.5%	3,197
R_6	216	80.7%	79.3%	3,260
R_7	217	81.1%	79.6%	3,329
R_8	218	81.5%	80.0%	3,588

Findings are raw counts on the pinned latest snapshot, not correctness labels

Table 3: Nanobot and Hermes trajectories

Target	Round	Rules	Patch Recall	Success	Findings
Nanobot	R_0	136	77.3%	22.7%	5,017
	R_1	136	86.4%	54.5%	5,009
	R_2	136	90.9%	68.2%	5,066
Hermes	R_0	173	75.0%	33.3%	92,963
	R_1	172	75.0%	50.0%	92,924
	R_2	172	83.3%	58.3%	92,981

All 14 new rules were added by R_3; later accepted changes modified existing rules, while the residual frontier contracted from 109 to 52 cases. Success rose another 3.6 points after additions stopped, indicating value from re-grounding as well as adding rules. The run ended at its eight-round budget with residuals remaining, not at semantic saturation. Latest-snapshot findings rose from 1,001 to 3,588; because Success is repair-local, this growth does not bear on precision. Iterative semantic revision fits 80.0% of the construction repairs under Success, with most gains in the first three rounds.

5.3 RQ2: CROSS-RUNTIME ADAPTATION

Each target R_0 was frozen before target vulnerability/fix feedback. Nanobot kept 40 of 180 source rules unchanged, modified 125, culled 14, and one rule was excluded because its adaptation record was incomplete; Hermes kept 23, modified 150, and culled 7. Rules passing realizability and validity checks formed R_0: 136 for Nanobot and 173 for Hermes. Nanobot localized 17/22 repair regions (77.3%) but achieved Success on 5/22 (22.7%); Hermes localized 9/12 (75.0%) but succeeded on 4/12 (33.3%). Thus the adapted corpora often reached repair locations but rarely distinguished vulnerable from fixed code. Hermes's median vulnerable/fixed finding-set Jaccard was 0.9902, and post-hoc attribution found only one of its four successes to be adapted transfer rather than incidental patch overlap. Feedback-free adaptation transfers localization substantially better than discrimination.

5.4 RQ3: TARGET-SPECIFIC REFINEMENT

As shown in Table 3, two refinement rounds driven by target historical vulnerability/fix feedback raised Nanobot Success from 22.7% to 68.2% and Patch Recall from 77.3% to 90.9%, and Hermes Success from 33.3% to 58.3% and Patch Recall from 75.0% to 83.3%.

Nanobot remained at 136 rules, Hermes changed from 173 to 172, and latest-snapshot finding counts stayed close to R_0, so the gain is not explained by more rules or findings. Refinement changed grounding rather than adding rules; Nanobot's final analysis found no missing AUGMENT hypothesis (Appendix B). Because no matched target-from-scratch baseline is available, these results establish the benefit of refinement to the adapted corpora, not an advantage over synthesizing target corpora from scratch.

Hermes gained less. Its finding volume, roughly 18 times Nanobot's, remained near R_0, consistent with broad queries; three `BACKEND_LIMITED` residuals also expose executable limits. On these small refinement sets, target feedback improves discrimination by re-grounding adapted rules, subject to backend limits.

5.5 Ablation and Baseline Comparison

On the same 275 OpenClaw cases, we compare LARA with existing rule sets and a query synthesizer, and ablate its components to isolate the source of its gains.

Table 4: Baselines (275 construction cases)

Method	Rules/queries	Patch Recall	Success
Stock Semgrep	273	3.3%	1.8%
Stock CodeQL	71	2.9%	0.4%
QLCoder	288	52.7%	40.7%
LARA	218	**81.5%**	**80.0%**

Table 5: Ablation; Δ: cases gained over R_0

Variant	Queries	Patch Recall	Success	Δ
w/o fuzzing (R_0)	166	61.1%	61.1%	—
w/o sem. mutation	168	62.2%	62.2%	+3
Full LARA (R_8)	218	**81.5%**	**80.0%**	+52

Stock Semgrep and CodeQL reach 1.8% and 0.4% Success on these domain-specific repairs. QLCoder reaches 40.7% by synthesizing a query from each case's own vulnerable/fixed pair, whereas LARA searches a shared corpus across all construction cases. The methods therefore use different units of synthesis and the comparison is contextual, not a controlled claim of superiority.

Table 5 shows where the gain comes from. Seed synthesis alone reaches 61.1%, and iteration recovers 52 more cases. With the same eight rounds of residual feedback, executable refinement, and deterministic validation but its 166 rules frozen, *w/o semantic mutation* gains only three. Query refinement changes how a rule is matched, not what it asserts, so it cannot fix a wrong entity, effect, relation, or scope, or a missing rule. Full LARA revises the rules themselves through CULL, MODIFY, and AUGMENT, growing from 166 to 180 rules; 49 of its 52 gains (94.2%) require semantic mutation. The gain therefore comes mainly from searching the space of semantic rules and only marginally from optimizing their CodeQL queries.

6 Latest-Snapshot Auditing and Verification

6.1 Auditing and Verification Protocol

LARA executes frozen corpora on pinned latest snapshots. OpenHands is deployment-only: it receives the Nanobot corpus unchanged, without OpenHands history or adaptation. Its local execution mode exposes the same privileged host operations as the other runtimes. Latest-snapshot findings never update a corpus and, because these revisions have no vulnerability labels, require contextual review and final human judgment.

Outputs were deduplicated to distinct (query, file, line) triples and grouped by code location into root causes. Two review sets served different purposes. First, an unweighted simple random sample of 1,035 production-code root causes was screened with a uniform open-audit instrument, designed for 90% confidence and a five-point target margin of error. Second, an enriched deep-review set contained 501 root causes: 480 selected through targeted review and 21 added only through random sampling. The latter is not a probability sample and cannot estimate a population rate.

LLM-assisted review tested rule claims, used findings as location pointers for open auditing, and examined enclosing functions, callers, guards, trust boundaries, and relevant state for surviving candidates. The LLM supplied triage and contextual reasoning, including PoC recipes, but no ground-truth label. Human researchers made the final security judgments and authored and executed local PoCs where applicable.

6.2 Latest-Snapshot Findings

In the random sample, 66 of 1,035 root causes were labeled potential security candidates: 6.38% (Wilson 90% CI: 5.24–7.74%). This rate measures screening yield: roughly one in sixteen sampled root causes warranted further attention. It estimates neither vulnerability prevalence nor analyzer precision.

Across the two review sets, 32 candidates entered confirmation review: 20 originating in the random sample and 12 in targeted review. One code-evidence-only record is excluded; of 31 candidates with executed, researcher-authored PoCs, 25 held under their stated preconditions and six were falsified.

We report these 25 as *PoC-validated vulnerabilities*; responsible disclosure (including CVE-filing and maintainer-confirmation) process is ongoing. OpenHands contributed 11 while using the Nanobot corpus unchanged, showing that the frozen rules provided useful audit locations in an independently implemented runtime; the count depends on review effort and is not a per-project rate. Appendix C gives the inventory.

Representative candidates include an OpenClaw session identifier parsed as a subprocess flag, a Nanobot media reference that read a local file for outbound sending, a Hermes adapter that read a `file://` path before its URL guard, and an OpenHands upload endpoint accepting an absolute destination path.

6.3 Validation and Falsified Candidates

The six falsified PoCs exposed guards, defaults, provenance constraints, or execution order that defeated the hypothesized effect. The code-evidence-only record and all reviewed candidates without individual PoCs remain outside the reported set. Frozen corpora can therefore provide audit locations, but static findings and LLM-assisted triage are not vulnerability verdicts.

7 Discussion and Limitations

The experiments support three bounded conclusions. First, residual-guided semantic revision improves fit to the OpenClaw construction cases beyond query-only refinement. Second, feedback-free adaptation often localizes target repairs but transfers little discrimination; target-specific refinement improves the latter on the same small histories. Third, frozen rules can identify locations worth human investigation on current code, including an unadapted runtime.

What transfers. The gap between target R_0 localization and Success clarifies the level of reuse. A rule can identify a component and protected effect across different APIs, but not necessarily the predicate separating insecure from repaired behavior. Refinement supplies that discrimination by changing the target binding. Transfer therefore provides a grounding scaffold, not an out-of-the-box detector.

How semantic revision changes an executable rule. A Nanobot sender-admission query selected a call that remained on both sides of two repairs, localizing but discriminating neither. A MODIFY regrounded the rule around the unattested identity claim and protected messaging effect; the regenerated query achieved Success on both cases without project-specific symbols or lost successes. Query-only refinement cannot repair a wrong entity, effect, relation, or scope.

Intended use. LARA emits review locations. Contextual screening rejects hypotheses defeated by program context, and an executed PoC supports the remaining claim. The random sample estimates screening yield while the enriched set broadens coverage; neither estimates deployment false-positive rate. LARA therefore prioritizes human audit.

OpenClaw's 275 cases are construction data, and final Nanobot/Hermes scores use their refinement cases; none is a held-out estimate of generalization. The small target corpora and absence of a matched target-from-scratch baseline further limit transfer claims. Patch Recall and Success are existential repair-region metrics that ignore findings elsewhere, so false-positive rate and deployment precision remain unmeasured. Hermes exposes broad queries and backend limits. Semantic saturation is operational rather than a completeness guarantee, and proposal stability across models or stochastic reruns remains untested.

8 Conclusion

LARA formulates executable-rule synthesis as verifier-guided, rule-level semantic fuzzing. Rules transfer enough structure to localize cross-runtime repairs, but target grounding supplies discrimination. Frozen-corpus review yielded 25 PoC-validated candidates, supporting LARA as an audit aid rather than an automated verdict.

Ethics Statement

This work studies vulnerabilities in open-source agent runtimes. All proof-of-concept exploits were authored by the researchers and executed only locally, against pinned revisions on researcher-controlled machines; no deployed service or third-party user was targeted. We are reporting all 25 PoC-validated candidates to the respective developers through their security disclosure channels. The paper describes each candidate only at the level of its mechanism and preconditions and includes no exploit code; PoC code and further candidate details will be released only after the corresponding reports are resolved. The study involves no human subjects or personal data.

Reproducibility Statement

Sections 3–4 and Appendix B describe the rule representation, the semantic action classes, the deterministic acceptance rule, and the residual frontier; Section 5 defines the metrics and the freeze protocol. Every frozen corpus is recorded with a manifest containing the corpus hash, per-query hashes, the benchmark hash, the backend module hash, and the CodeQL version. The frozen corpora and evaluation artifacts are available at an anonymous repository: https://anonymous-hf.com/a/7mb5mgogdeg2/. PoC code for candidates still under disclosure is withheld until the corresponding reports are resolved.

AI Use Statement

LLMs are a component of the method. In LARA's rule synthesizer, an LLM (Claude Opus 5) proposes seed rules, residual groupings, semantic revisions (CULL, MODIFY, AUGMENT), and target groundings, and, after each target R0 was frozen, assisted in mining target historical cases used for target-specific refinement; every proposal is lowered to CodeQL and accepted or rejected by deterministic compilation, execution, and validation (Section 4). In the latest-snapshot study, an LLM assisted triage and contextual review of findings and drafted PoC recipes; researchers made all final security judgments and authored and executed the PoCs (Section 6). LLM assistants were also used for code editing, language editing and restructuring of the manuscript, and drafting figure layouts. The authors reviewed and verified all LLM-assisted content and take full responsibility for the paper.

References

Avi Arora, Jinu Jang, and Roshanak Zilouchian Moghaddam. SetupBench: Assessing software engineering agents' ability to bootstrap development environments. *arXiv preprint arXiv:2507.09063*, 2025.

Pavel Avgustinov, Oege de Moor, Michael Peyton Jones, and Max Schäfer. QL: Object-oriented queries on relational data. In *30th European Conference on Object-Oriented Programming (ECOOP 2016)*, volume 56 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pp. 2:1–2:25. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2016. doi: 10.4230/LIPIcs.ECOOP.2016.2.

Luca Beurer-Kellner, Aleksei Kudrinskii, Marco Milanta, Kristian Bonde Nielsen, Hemang Sarkar, and Liran Tal. 280+ leaky skills: How openclaw & clawhub are exposing api keys and pii. *Snyk Blog, February*, 2026a.

Luca Beurer-Kellner, Aleksei Kudrinskii, Marco Milanta, Kristian Bonde Nielsen, Hemang Sarkar, and Liran Tal. Technical report: Exploring the emerging threats of the agent skill ecosystem. *arXiv preprint arXiv:2605.28588*, 2026b.

Luca Beurer-Kellner et al. ToxicSkills: Snyk Finds Malware and Prompt Injection in 36% of AI Agent Skills. Snyk Blog, February 2026c. https://snyk.io/blog/toxicskills-malicious-ai-agent-skills-clawhub. Accessed 2026-06-17.

Hwan Chang, Yonghyun Jun, and Hwanhee Lee. Chatinject: Abusing chat templates for prompt injection in llm agents. *arXiv preprint arXiv:2509.22830*, 2025.

Pei Chen, Baichao An, Mengying Wu, Binwang Wan, Geng Hong, Jinsong Chen, Xudong Pan, Jiarun Dai, and Min Yang. Rethinking mcp security: A large-scale study of runtime mcp servers and security scanner reliability. *arXiv preprint arXiv:2607.11086*, 2026.

Edoardo Debenedetti, Jie Zhang, Mislav Balunovic, Luca Beurer-Kellner, Marc Fischer, and Florian Tramèr. Agentdojo: A dynamic environment to evaluate prompt injection attacks and defenses for llm agents. *Advances in Neural Information Processing Systems*, 37:82895–82920, 2024.

Randall Degges. Securing the Agent Skill Ecosystem: How Snyk and Vercel Are Protecting Developers. Snyk Blog, February 2026. https://snyk.io/blog/snyk-vercel-securing-agent-skill-ecosystem/. Accessed 2026-06-17.

GitHub. CodeQL: Semantic code analysis engine. https://codeql.github.com, 2025. Accessed: 2026-06-16.

Kai Greshake, Sahar Abdelnabi, Shailesh Mishra, Christoph Endres, Thorsten Holz, and Mario Fritz. Not what you've signed up for: Compromising real-world llm-integrated applications with indirect prompt injection. In *Proceedings of the 16th ACM workshop on artificial intelligence and security*, pp. 79–90, 2023.

Muhammad Hamza, Muhammad Azeem Akbar, Wardah Naeem Awan, and Muhammad Shoaib. Understanding security issues in open-source agentic ai frameworks: An empirical analysis. *Available at SSRN 6627158*, 2026.

Mohammed Mehedi Hasan, Hao Li, Emad Fallahzadeh, Gopi Krishnan Rajbahadur, Bram Adams, and Ahmed E Hassan. Model context protocol (mcp) at first glance: Studying the security and maintainability of mcp servers. *arXiv preprint arXiv:2506.13538*, 2025.

HKUDS. Nanobot: Lightweight, open-source AI agent for tools, chats, and workflows. https://github.com/HKUDS/nanobot, 2026. Accessed: 2026-06-16.

Koi Security. ClawHavoc: 341 malicious clawed skills found by the bot they were targeting, 2026. Koi Security report (accessed 2026-05-11).

Pratyay Kumar, Miguel Antonio Guirao Aguilera, Srikathyayani Srikanteswara, Satyajayant Misra, and Abu Saleh Md Tayeen. Mcp-in-sos: Risk assessment framework for open-source mcp servers. *arXiv preprint arXiv:2603.10194*, 2026.

Ziyang Li, Saikat Dutta, and Mayur Naik. IRIS: LLM-Assisted Static Analysis for Detecting Security Vulnerabilities. In *International Conference on Learning Representations*, 2025. URL https://arxiv.org/abs/2405.17238.

Yi Liu, Zhihao Chen, Yanjun Zhang, Gelei Deng, Yuekang Li, Jianting Ning, Ying Zhang, and Leo Yu Zhang. Malicious agent skills in the wild: A large-scale security empirical study. *arXiv preprint arXiv:2602.06547*, 2026a.

Yi Liu, Weizhe Wang, Ruitao Feng, Yao Zhang, Guangquan Xu, Gelei Deng, Yuekang Li, and Leo Zhang. Agent skills in the wild: An empirical study of security vulnerabilities at scale. *arXiv preprint arXiv:2601.10338*, 2026b.

Wanlun Ma, Qing-Long Han, Xiaogang Zhu, Wei Zhou, Junwu Xiong, Zhihang Ren, Sheng Wen, and Yang Xiang. Openclaw in the wild: Security analysis of autonomous agents. *IEEE/CAA Journal of Automatica Sinica*, 13(6):1257–1273, 2026.

Microsoft. The STRIDE threat model. https://learn.microsoft.com/en-us/previous-versions/commerce-server/ee823878(v=cs.20), 2009. Accessed: 2026-06-16.

Yu Nong, Haipeng Cai, Pengfei Ye, Li Li, and Feng Chen. Evaluating and comparing memory error vulnerability detectors. *Information and Software Technology*, 137:106614, 2021.

Yu Nong, Rainy Sharma, Abdelwahab Hamou-Lhadj, Xiapu Luo, and Haipeng Cai. Open science in software engineering: A study on deep learning-based vulnerability detection. *IEEE Transactions on Software Engineering*, 49(4):1983–2005, 2022.

Rupam Patir, Keyan Guo, Suvadra Barua, Abhijeet Pathak, Dinesh Gudimetla, Jiawei Guo, Hongxin Hu, and Haipeng Cai. DualGauge: Automated joint security-functionality benchmarking of specification-only code generation by LLMs and coding agents. *arXiv preprint arXiv:2511.20709*, 2026.

Semgrep, Inc. Semgrep: Lightweight static analysis for many languages. https://semgrep.dev, 2025. Accessed: 2026-06-16.

Adam Shostack. *Threat Modeling: Designing for Security*. John Wiley & Sons, Indianapolis, IN, 2014. ISBN 978-1-118-80999-0.

Snyk. agent-scan: Security Scanner for AI Agents, MCP Servers, and Agent Skills. GitHub repository, 2026. https://github.com/snyk/agent-scan. Accessed 2026-06-17.

Peter Steinberger and OpenClaw Contributors. OpenClaw: Personal AI assistant. https://github.com/openclaw/openclaw, 2026. Accessed: 2026-06-16.

Surada Suwansathit, Yuxuan Zhang, and Guofei Gu. A security analysis of the openclaw ai agent framework. *arXiv preprint arXiv:2603.27517*, 2026.

Claire Wang, Ziyang Li, Saikat Dutta, and Mayur Naik. Qlcoder: A query synthesizer for static analysis of security vulnerabilities. In *International Conference on Learning Representations*, volume 2026, pp. 25205–25241, 2026.

Qiusi Zhan, Zhixiang Liang, Zifan Ying, and Daniel Kang. Injecagent: Benchmarking indirect prompt injections in tool-integrated large language model agents. In *Findings of the Association for Computational Linguistics: ACL 2024*, pp. 10471–10506, 2024.

Qiusi Zhan, Richard Fang, Henil Shalin Panchal, and Daniel Kang. Adaptive attacks break defenses against indirect prompt injection attacks on llm agents. In *Findings of the Association for Computational Linguistics: NAACL 2025*, pp. 7101–7117, 2025.

Haiyue Zhang, Yi Nian, and Yue Zhao. Agent audit: A security analysis system for llm agent applications. *arXiv preprint arXiv:2603.22853*, 2026.

A Expanded Related Work

A.1 Prompt Injection and Behavioral Agent Evaluation

A substantial line of work studies how LLM agents can be manipulated through their inputs. Greshake et al. (Greshake et al., 2023) introduced indirect prompt injection, in which adversarial instructions hidden in retrieved documents or tool outputs propagate into the prompt context. Subsequent studies characterize adaptive and chat-template-based attacks (Zhan et al., 2025; Chang et al., 2025), while behavioral benchmarks evaluate agents end to end through their inputs and actions (Arora et al., 2025; Patir et al., 2026). This literature provides attack models and evaluation methods for measuring agent behavior under adversarial input.

A.2 Security of Agent Tools, Skills, and Marketplaces

Another body of work examines third-party artifacts used by agents. Marketplace investigations document malicious skills and credential exposure in ecosystems such as ClawHub (Koi Security, 2026; Beurer-Kellner et al., 2026a). Large-scale empirical studies analyze vulnerabilities and malicious behavior across agent skills (Liu et al., 2026b;a), while technical reports and industry tools characterize emerging skill-ecosystem threats and provide scanners for agents, MCP servers, and skills (Beurer-Kellner et al., 2026b;c; Degges, 2026; Snyk, 2026). Together, these studies describe the risks introduced when agents acquire and execute third-party capabilities.

A.3 Empirical Studies of Agent Frameworks

Recent work also analyzes agent applications and open-source frameworks. Agent Audit presents a source-code security-analysis pipeline for LLM-agent applications (Zhang et al., 2026). OpenClaw in the Wild applies a trust-boundary-first analysis to autonomous agents (Ma et al., 2026), and A Security Analysis of the OpenClaw AI Agent Framework develops a taxonomy of OpenClaw bugs (Suwansathit et al., 2026). A broader empirical study categorizes security issues reported across open-source agentic AI frameworks (Hamza et al., 2026). These efforts provide complementary system-level and empirical views of security weaknesses in deployed agent software.

A.4 Security of MCP Servers

The Model Context Protocol has motivated several large-scale studies of servers that connect agents to external tools and data. Hasan et al. (2025) combine security and maintainability analyses in an audit of 1,899 MCP servers. MCP-in-SoS proposes a risk-assessment framework for open-source MCP servers (Kumar et al., 2026), while a large-scale runtime study dynamically evaluates MCP servers and examines the reliability of security scanners (Chen et al., 2026). This work covers repository-level properties, structured risk assessment, and runtime behavior in the MCP ecosystem.

A.5 Static Analysis for Vulnerability Detection

Static vulnerability analysis is a mature field (Nong et al., 2021). Semgrep encodes syntactic templates with metavariable constraints (Nong et al., 2022; Semgrep, Inc., 2025), while CodeQL evaluates declarative structural and data-flow queries (Avgustinov et al., 2016; GitHub, 2025). IRIS combines LLM-inferred taint specifications and contextual triage with CodeQL (Li et al., 2025); QLCoder synthesizes a query for each vulnerability through LLM and execution feedback (Wang et al., 2026). LARA instead searches over an intermediate corpus of semantic requirements intended to be shared across cases and re-grounded across runtimes; its baseline comparison with QLCoder is contextual because the synthesis units differ.

B Additional LARA Method Details

Representation and records. Each rule records applicability and scope, a security-sensitive entity, a protected effect, their required relation, target bindings, query realizations, and lineage. Relation tags include `MUST_DOMINATE` and `MUST_FAIL_CLOSED`. Existing Semgrep and CodeQL rules were normalized with historical-fix hypotheses. Rule-level provenance is retained, although aggregate per-source counts are unavailable. The resulting S_0 held 166 rules, each implemented by one CodeQL query. Frozen evaluation uses CodeQL. Round snapshots and query/corpus hashes are recorded. Residual states are `ACTIONABLE`, `EXHAUSTED`, `BACKEND_LIMITED`, and `SPECIALIZATION_ONLY`; an inexpressible proposal is `UNREALIZABLE_CURRENT_BACKEND`.

Residual frontier and acceptance. The residual frontier of a round contains the construction cases for which no rule in the current corpus passes the verifier's reconstruction check, a deterministic predicate over a query's vulnerable-revision findings and the case's repair region. The frontier held 109 cases after the seed round and 52 after the eighth round. Reconstruction is stricter than Patch Recall and differs from Success (after the eighth round, 223 cases were reconstructed, 224 localized, and 220 successful), so the evaluation metrics of Section 5 do not determine the frontier. A proposal is accepted only if its query compiles, reconstructs and discriminates at least one residual case that motivated it, and loses no existing success or recall.

Illustrative residual-guided revision. In Nanobot, CVE-2026-2577 and CVE-2026-33654 shared sender-admission residuals. Their parent rule's query selected an admission *call* that survived the fix and fired on both revisions. A MODIFY re-grounded the rule around the *unattested identity claim* and adjusted the witness/effect relation. Deterministic validation found new Success on both cases, no lost successes, and no project-specific symbols; one revised rule served two independent implementations.

Recorded residual frontiers. OpenClaw accepted no AUGMENT after R_3; later residuals clustered around localization and query-expressiveness limits, and the run hit its eight-round ceiling. Nanobot's

final analysis found no missing AUGMENT hypothesis after probing source rules, with lower-yield MODIFY proposals. Hermes froze at R_2 with two `ACTIONABLE` and three `BACKEND_LIMITED` residuals: its frontier was *not* empty.

Evaluation metrics. For case c in the evaluated corpus $\mathcal{C}$, let $\mathcal{V}_c$ be its valid queries, P_c^v and P_c^f the corresponding vulnerable and fixed repair regions, and $F_{c,r}^v$ and $F_{c,r}^f$ the findings of query r on each revision. Repair regions are fix hunks padded by three lines. We compute

$$\text{Patch Recall} = \frac{1}{|\mathcal{C}|} \sum_{c \in \mathcal{C}} \mathbf{1}\left[\exists r \in \mathcal{V}_c : \; F_{c,r}^v \cap P_c^v \neq \varnothing\right], \tag{1}$$

$$\text{Success} = \frac{1}{|\mathcal{C}|} \sum_{c \in \mathcal{C}} \mathbf{1}\left[\exists r \in \mathcal{V}_c : \; F_{c,r}^v \cap P_c^v \neq \varnothing \;\wedge\; F_{c,r}^f \cap P_c^f = \varnothing\right]. \tag{2}$$

C REPORTED LATEST-SNAPSHOT CANDIDATES

Each of the 25 candidates below has an executed, researcher-authored local PoC. They are previously unreported vulnerability candidates, not maintainer-confirmed vulnerabilities.

Table 6: Latest-snapshot review. SRS is the screening sample; the enriched deep-review set is not a probability sample. Reported candidates have executed PoCs.

Project	Frozen corpus used	SRS screened	Deep-review set	PoC-supported
OpenClaw	OpenClaw S_{final}	253	134	2
Nanobot	Nanobot R_2	257	114	8
Hermes	Hermes R_2	270	143	4
OpenHands	Nanobot R_2 unchanged	255	110	11
Total	—	1,035	501	25

Table 7: Reported latest-snapshot candidates 1–13 of 25.

#	Project	Source location and reported defect
1	OpenClaw	`extensions/acpx/src/runtime-internals/mcp-proxy.mjs:121`: secret-in-process-argv (CWE-214 / CWE-522)
2	OpenClaw	`extensions/codex/src/node-cli-sessions.ts:265`: argument-injection
3	Nanobot	`nanobot/agent/tools/shell.py:823`: path-traversal (incomplete canonicalization in a declared confinement boundary)
4	Nanobot	`nanobot/channels/weixin/runtime.py:1502`: ssrf
5	Nanobot	`nanobot/channels/email/runtime.py:853`: unhandled-exception-in-error-path (availability)
6	Nanobot	`nanobot/channels/qq/runtime.py:465`: path-traversal / arbitrary-local-file-read (exfiltration)
7	Nanobot	`nanobot/apps/cli/service.py:1368`: path-traversal / symlink-following
8	Nanobot	`nanobot/agent/tools/shell.py:459`: fail-open security control
9	Nanobot	`nanobot/apps/cli/service.py:385`: path-traversal (separator confusion)
10	Nanobot	`nanobot/agent/tools/web.py:89`: regex-complexity DoS
11	Hermes	`plugins/platforms/mattermost/adapter.py:376`: local-file-exfiltration (scheme dispatch above the URL guard)
12	Hermes	`gateway/platforms/yuanbao.py:1370`: credential-leak-on-redirect
13	Hermes	`tools/mcp_tool_errors.py:291`: credential-leak-on-redirect (insecure default)

Table 8: Reported latest-snapshot candidates 14–25 of 25.

#	Project	Source location and reported defect
14	Hermes	`plugins/platforms/telegram/adapter.py:4830`: html-injection / encoding inconsistency
15	OpenHands	`openhands-sdk/openhands/sdk/conversation/impl/remote_conversation.py:1180`: remote-supplied filename path traversal write
16	OpenHands	`openhands-agent-server/openhands/agent_server/sockets.py:154`: missing-connection-provenance-check (cross-site WebSocket hijacking)
17	OpenHands	`openhands-sdk/openhands/sdk/hooks/executor.py:505`: resource-exhaustion/liveness (pipe-buffer deadlock)
18	OpenHands	`openhands-agent-server/openhands/agent_server/file_router.py:93`: unrestricted-file-write / missing path containment
19	OpenHands	`openhands-agent-server/openhands/agent_server/workspace_router.py:90`: path-traversal via symlink following
20	OpenHands	`openhands-sdk/openhands/sdk/hooks/executor.py:579`: error-handling-fails-open / security-decision-downgrade
21	OpenHands	`openhands-sdk/openhands/sdk/io/local.py:71`: stale-cache / missing cache invalidation
22	OpenHands	`openhands-sdk/openhands/sdk/skills/utils.py:367`: path-traversal / symlink-following (information disclosure)
23	OpenHands	`openhands-sdk/openhands/sdk/utils/command.py:146`: broken timeout enforcement (fail-open watchdog)
24	OpenHands	`openhands-sdk/openhands/sdk/conversation/impl/local_conversation.py:1112`: untrusted-code loading / missing workspace trust decision
25	OpenHands	`scripts/render_examples_report.py:223`: markdown/table injection (output encoding)

D Study Setup and Prompt Templates

Model and decoding. As shown in Table 9, LARA's LLM layer uses Claude Opus 5 with adaptive thinking. Sampling parameters are left at their API defaults, and determinism is obtained from the deterministic components rather than from a seed.

Table 9: Model and decoding settings

Setting	Value
Model	Claude Opus 5 (`claude-opus-5`)
Thinking	adaptive
Temperature	not set (the model does not accept the parameter)
Top-p / top-k	not set
Seed	not available through the API
Concurrency	12 parallel workers

Token budgets and usage. The study has no separate context-window setting: each request's input is the rendered template plus its evidence payload, and adaptive thinking counts against the output budget. Output ceilings were raised twice during the study after the hardest cases exhausted the budget while thinking and returned empty text; the prompts and the model were unchanged. The largest request carried 117,110 input tokens and the largest response 73,015 output tokens; no input was truncated.

Deterministic components. As shown in Table 10, compilation, execution, and scoring use fixed tool versions and limits. CodeQL decides the relations `MUST_DOMINATE`, `MUST_FAIL_CLOSED`, `MUST_REMAIN_WITHIN`, `MUST_AGREE`, `MUST_PRESERVE_AUTHORITY`, `MUST_PROPAGATE_SECURITY_STATE`, and `MUST_NOT_WEAKEN_ENFORCEMENT`; Semgrep decides

MUST_FAIL_CLOSED, MUST_NOT_WEAKEN_ENFORCEMENT, and MUST_DOMINATE for local patterns only. Each rule is grounded once against a reference revision; other revisions reuse the cached symbol binding without a model call, and a rule with no usable binding on a revision is recorded as not applicable there.

Table 10: Deterministic components

Component	Version or setting
CodeQL	2.23.5
Semgrep	1.177.0
CodeQL threads / RAM	8 threads, 12,000 MB
Per-query timeout	600 s
Database build timeout	5,400 s
Patch-region padding	3 lines on each side of a hunk

Selected prompt templates. The templates below are reproduced verbatim as rendered by the template engine, except that em dashes are typeset as "–". {{field}} markers are replaced with case evidence at call time; nothing else varies between calls. The templates predate the paper's terminology: a *family* or *invariant* corresponds to a rule, a *realization* or *detector specification* to its query, and *migration* to cross-runtime adaptation.

vulnerability_characterization Phase I: characterizing one historical vulnerability.

```
# Vulnerability characterization v1 -- independent reading of one historical vulnerability

You are analysing ONE real historical vulnerability of a local LLM-agent runtime and its
     repair.
Read the evidence below and characterize the vulnerability. Do NOT yet propose a detection
     rule, a generalized rule, or any taxonomy label.

Use only generic security and program-analysis vocabulary, for example: caller, principal,
     identity, source, sink, guard, authorization,
authentication, resource, privileged effect, state, dataflow, control flow, dominance,
     provenance, wrapper, middleware, representation,
canonicalization, execution boundary. Do not invent or use project-specific category codes.

Base every statement on the evidence. Where the evidence is insufficient, say so instead of
     guessing.

## Evidence
Advisory: {{advisory}}
Changed files: {{fix_files}}
Vulnerable implementation (fix regions): {{vulnerable_code}}
Fixed implementation (fix regions): {{fixed_code}}
Patch (unified diff): {{patch}}
Deterministic runtime-map facts for the vulnerable revision (entity kinds, symbols, files,
     enclosing functions, wrappers, mediation): {{runtime_facts}}

## Output
Exactly one JSON object, no prose outside it:
{
 "case_id": "{{case_id}}",
 "security_property_violated": "<what must hold and did not>",
 "security_boundary": "<which trust boundary is crossed>",
 "trusted_entities": ["..."],
 "untrusted_entities": ["..."],
 "privileged_effect_or_protected_resource": "<the effect or resource at stake>",
 "failure_mechanism": "<how the property is broken, in mechanism terms>",
 "repair_mechanism": "<what the fix changed, in mechanism terms>",
 "vulnerable_side_entities": {"<role-ish name>": "<concrete symbol/file/expression on the
     vulnerable revision>"},
 "fixed_side_entities": {"<role-ish name>": "<concrete symbol/file/expression on the fixed
     revision>"},
 "required_control_data_or_provenance_relationship": "<the relationship a checker must
     verify>",
 "implementation_specific_details": ["<things that are incidental to this codebase>"],
 "potentially_reusable_semantics": ["<what could plausibly recur in other systems>"],
 "cwe_ids": ["CWE-nnn", "..."],
 "evidence_sufficient": true|false,
 "evidence_gaps": ["..."]
}
```

semantic_validation Phase I: semantic validation of a candidate rule.

```
# Semantic validation v1 -- ACCEPT / CORRECT / REJECT / ABSTAIN

You validate a candidate semantic security rule for a local LLM-agent runtime. Coverage of
     any benchmark is NOT a criterion.

Frozen role vocabulary: {{roles}}
Frozen relations: {{relations}}
Candidate rule: {{rule}}
Origin: {{origin}}  (SEED from historical evidence / NORMALIZE from an existing rule /
     MODIFY|AUGMENT from a residual-frontier explanation / GROUND onto a target)
Evidence the rule claims to rest on: {{evidence}}
Parent rule (if a mutation): {{parent}}
Existing corpus rules with the same relation (for redundancy): {{siblings}}
Available deterministic backends: {{backends}}

Answer each question with a verdict and one sentence:
1. Is the security requirement meaningful (a real security property, not a code smell)?
2. Is it grounded in the evidence given (the evidence actually exhibits this property
     failing/holding)?
3. Does it preserve the intended security meaning (for mutations/groundings: same invariant
     as the parent)?
4. Are the roles appropriate for this relation's signature?
5. Is the relation the right one for the mechanism?
6. Are the applicability assumptions supported by the runtime structure described?
7. Is it redundant with a listed sibling (same invariant, same roles, same qualifiers)?
8. Is there a realizable deterministic backend (which, and why)?
9. For mutations: did the mutation weaken or distort the invariant?

Then decide: ACCEPT (valid as is), CORRECT (valid after a specific correction you provide --
      role, relation, wording, qualifier, applicability; never add target-specific names),
     REJECT (invalid / not a security requirement / distorted), ABSTAIN (cannot judge from
     the evidence; state what is missing).

Output exactly one JSON object:
{"outcome": "ACCEPT|CORRECT|REJECT|ABSTAIN", "answers": {"1": "...", ..., "9": "..."}, "
     corrected_rule": <Rule IR object or null>, "redundant_with": "<rule_id or null>", "
     backend": "codeql|semgrep|both|BACKEND_UNSUPPORTED", "rationale": "<two sentences>"}
```

semantic_mutation Phase I: CULL/MODIFY/AUGMENT from a residual explanation.

```
# Semantic mutation v1 -- MODIFY / AUGMENT / CULL from a residual-frontier explanation

Produce concrete semantic rule mutations that follow from the explanation below. MODIFY/
     AUGMENT/CULL are action classes, not syntactic operators:
derive the change from the security semantics in the explanation. Preserve the security
     requirement under MODIFY; create a new invariant under AUGMENT;
CULL only for semantic invalidity, genuine inapplicability or redundancy (zero findings is
     never a reason).

Frozen role vocabulary: {{roles}}
Frozen relations: {{relations}}
Explanation (reasoning_ref = {{reasoning_ref}}): {{explanation}}
Rules targeted (full IR): {{target_rules}}
Deterministic evidence available to the realization (runtime-map facts kinds: {{fact_kinds
     }}): {{evidence_summary}}

MODIFY may change: role realization, source/sink realization, scope, wrapper/middleware/
     framework mediation qualifier, representation, context,
authority-provenance qualifier, dataflow/interprocedural qualifier, temporal stage,
     implementation family. It must NOT change the invariant.
AUGMENT adds a new rule for a missing requirement / relation realization / mechanism. It
     must be reusable, not a description of one case.
CULL marks a rule CULLED with a semantic justification.

Constraints: no target-specific symbol names in requirement or role abstractions; bindings
     stay null (grounding happens later) unless the
explanation concerns a binding family expressible in generic terms (then use {"family": "<
     runtime-map category>"}); every output carries
"reasoning_ref": "{{reasoning_ref}}" and "mutation_action".

Output exactly one JSON object: {"mutations": [{"mutation_action": "MODIFY|AUGMENT|CULL", "
     parent_rule_id": "<or null>", "reasoning_ref": "{{reasoning_ref}}",
 "what_changed": "<precise delta>", "why_preserves_requirement": "<MODIFY only>", "rule": <
     Rule IR object or {"rule_id": ..., "status": "CULLED", "justification": ...}>}]}
Rule IR object:
{
 "rule_id": "<short slug>",
```

```
 "stride": "Spoofing|Tampering|Elevation of Privilege|Information Disclosure",
 "cat": "CAT-1..CAT-5",
 "requirement": "<one reusable security invariant, no target-specific names>",
 "relation": "<one of the 7 frozen relations>",
 "roles": {"<ROLE>": {"abstract": "<what plays this role, in general terms>", "binding":
     null}},
 "qualifiers": {"mediation": "direct|wrapper|middleware|framework", "provenance_sensitive":
     true|false, "interprocedural": true|false, "temporal": "<optional>", "representation":
     "<optional>", "local_syntactic": true|false},
 "scope": {"component": "<optional generic component kind, not a path>"},
 "applicability": ["<structural precondition in runtime-map vocabulary>"],
 "backend_hint": "codeql|semgrep|both",
 "rationale": {"violated_property": "...", "fix_changed": "...", "incidental_details":
     ["..."], "why_reusable": "..."}
}
```

`nanobot_family_migration` Phase II: KEEP/MODIFY/CULL adaptation of source rules.

```
# Migrate OpenClaw security families to Nanobot, at the invariant level

You are migrating security KNOWLEDGE, not code. For each OpenClaw semantic family below,
     decide whether its
underlying security property applies to Nanobot, and if so restate it in Nanobot's terms.

OpenClaw is a TypeScript agent runtime. Nanobot is a Python agent runtime. The transferred
     object is the
**security invariant** -- entity, protected effect, required relation. The executable
     realization is
target-specific and is NOT your concern here. An OpenClaw JavaScript API pattern rewritten
     as a Python API name
is **not** semantic migration and must not be what you produce.

## Nanobot at the v0.3.0 cutoff
{{architecture}}

You are given Nanobot's architecture, components, trust boundaries and the public symbol
     vocabulary of its
security-critical modules. You are NOT given any Nanobot vulnerability, patch or advisory,
     and you must not
speculate about specific historical bugs. Decide from the architecture alone whether the
     security surface exists.

## The OpenClaw families to migrate
{{families}}

## Decide, per family

- `KEEP` -- the same security-bearing entity, the same protected effect and the same
     required relation apply to
  Nanobot substantially unchanged. The implementation language differs; the security
     property does not.
- `MODIFY` -- the underlying security property transfers, but Nanobot expresses the entity,
     the effect, the trust
  boundary or the required relation differently, so the family must be adapted. Say exactly
     what differs.
- `CULL` -- Nanobot does not expose the relevant security surface, or the invariant is not
     meaningful for this
  runtime. Be willing to CULL: OpenClaw has integrations and subsystems Nanobot does not.

Judge by the triple, not the vocabulary. Two families can share a CWE and require different
     relations.

For `KEEP` and `MODIFY` restate all six fields. The Nanobot entity/effect/relation must be
     phrased so that a
Python realization could be written from them WITHOUT consulting any Nanobot patch: name the
      kind of construct,
the trust boundary it crosses, and the relation that must hold -- not a specific function
     that happens to be the
fix. If your Nanobot relation reads like a description of one known bug, you have gone too
     far.

`nanobot_grounding_hint` may name real Nanobot modules or symbols from the vocabulary above
     where they genuinely
carry the entity or effect (for example a path-confinement helper, an SSRF validator, the
     channel authorization
base). This is a binding aid, never the security content.

Return ONLY a JSON array, one object per family given, in a ```json fenced block:
```

```
[{
 "source_family_id": "<exactly as given>",
 "action": "KEEP|MODIFY|CULL",
 "source_entity": "<OpenClaw security-bearing entity>",
 "source_effect": "<OpenClaw protected effect>",
 "source_relation": "<OpenClaw required security relation>",
 "nanobot_entity": "<or null if CULL>",
 "nanobot_effect": "<or null if CULL>",
 "nanobot_relation": "<or null if CULL>",
 "nanobot_component": "<the Nanobot component(s) where this lives, or null>",
 "nanobot_trust_boundary": "<which boundary the entity crosses, or null>",
 "nanobot_grounding_hint": "<modules/symbols that could carry entity or effect, or null>",
 "what_differs": "<MODIFY only: precisely which of entity/effect/relation/boundary differs
     and how; null otherwise>",
 "rationale": "<two sentences: why this action, argued from the security triple>",
 "confidence": "HIGH|MEDIUM|LOW"
}]
```