Proving LLM-Found Authorization Bugs in AI-Generated Code

A Deterministic Oracle that Certifies Exploits with *No* LLM Verdict—and Refuses What It Cannot Prove

Minh Danh Ngo
Founder, VULX LTD
71–75 Shelton Street, London WC2H 9JQ, United Kingdom
contact@vulx.ai | vulx.ai

VulX Technical Whitepaper **v1.0** — June 20, 2026
For design partners and reviewers. Not peer-reviewed. Third-party targets anonymized (§12).

Abstract

Code is now written faster than it can be reviewed: a large and growing fraction of new software is generated by large language models (LLMs). The vulnerability class that benefits least from this speed-up, and suffers most from it, is *broken authorization*—IDOR / Broken Object-Level Authorization (BOLA)—because authorization is a property of an application's intended access *policy*, not a syntactic pattern a scanner can match. We argue that for AI-generated code the binding constraint is not detection but *discovery* (localizing where an access-control decision is missing relative to the declared policy) and *proof* (showing the gap is actually exploitable, without guessing). Concretely, this paper asks the question a security tool must answer before it can act on an LLM: *when a model claims an authorization bug exists, how do we know it is real—without trusting the model's word?*

We present **VulX**, whose central contribution is a *deterministic exploit oracle*. An LLM proposes; the oracle disposes. The ship-gate is a pure rule—`decide()` = `before.present` $\wedge \neg$ `after.present` $\wedge$ `control_ok_after`—and *never* an LLM verdict: a fix ships only when a real exploit is observed before the patch, is gone after it, and a benign request still succeeds (a mandatory negative control). The same rule makes the oracle a *precision gate*: when a proposed exploit does not actually fire at runtime, the oracle returns NOT_REPRODUCIBLE and *refuses* the finding rather than reporting it. Around this oracle, discovery and attack-authoring are autonomous (no hand-fed finding); the test-bed is harness-provisioned; and the fix is a reference fix used as a proof vehicle, not an autonomous fixer.

We report what is measured and label what is not. Across eleven real deployed applications, the oracle *confirmed* six broken-authorization exploits (each measured before-leak / after-fix / passing-control) and *declined* five more in three distinct ways—unproven impact, no measurable control, and an un-buildable target. Only one HIT was found *end-to-end* by autonomous discovery; the rest were proven on hand-fed shapes. Separately, a deterministic, LLM-free discovery floor recalls **5/6** of the proven shapes from source at high confidence (≥ 0.7; recall-first, with precision left to the oracle) and **2/2** on synthetic fixtures—advancing proof, breadth, and a discovery recall floor, though not yet end-to-end autonomous discovery at scale. The refusals—a tool declining real-looking findings—are the result we most want to show. As a baseline, two industry-standard static analyzers (Semgrep, CodeQL) flagged *none* of these authorization bugs—0/5 on the proven HITs; two controlled fixtures additionally exercise the full loop. We are explicit about scope: the evidence is small-n, early-stage, and—apart from one app—hand-fed; the reasoning model is hosted Claude Opus 4.8 (not a VulX-trained model); and autonomous

discovery of these shapes, cross-codebase variant analysis, and a trained attacker are roadmap. The thesis is not that an LLM can hack—others have shown that—but that its output can be made *trustworthy*: capability comes from the model; trust comes from the rule.

Honesty note. Tags [MEASURED] and [ROADMAP] appear throughout. All figures are *internal and self-reported* unless a primary external source is cited.

Keywords: broken access control; BOLA / IDOR; LLM security; deterministic exploit oracle; exploit verification; AI-generated code; Supabase / Postgres Row-Level Security.

Contents

1 Introduction

The economics of software creation changed when LLMs began writing production code. Speed went up; review did not. The result is a widening gap between code that *works* and code that has been *reasoned about for security.* Some of that gap is well served by existing tooling: hardcoded secrets, known-vulnerable dependencies, and unsafe primitives (`eval`, string-built SQL) are syntactic and a good scanner finds them.

Authorization is not like that. Whether a route is allowed to return an object depends on *who is asking* and on the application's *intended policy* ("a user may read only their own documents"). That policy lives across several artifacts—database Row-Level Security (RLS) rules, ownership columns and foreign keys, framework route guards—and the bug is the *absence* of a check relative to that policy. No pattern match expresses "this query should have been scoped to the caller and is not." This is precisely the class that AI-generated code gets wrong most often, because the model emits a plausible, working query and silently omits the ownership predicate.

The research question is narrow on purpose—not "can an LLM hack a site" (others have shown that it can), but *when an LLM claims a broken-access-control bug, how do we prove it is real—and decline it when it is not—without ever trusting the model's verdict?* This paper makes four claims, in order of importance:

1. **Correctness must come from a deterministic oracle, not a model** (§6, §10)—*this is the contribution.* A pure before/after/negative-control rule decides whether anything ships; no LLM renders the verdict, so model hallucination cannot produce a false "proven." The same rule makes the oracle a *precision gate*: a real-looking finding whose exploit does not fire is returned NOT_REPRODUCIBLE and *refused*—declining a false finding, not merely confirming a true one.
2. **Authorization is the dominant blind spot of AI-generated code** (§2). We motivate this with a public benchmark of real CVEs in AI-generated code and a real-world authorization CVE class.
3. **An LLM can autonomously discover and exploit these bugs** (§7, §8)—supporting, early-stage evidence, not a benchmarked rate. Opus 4.8 autonomously reconstructs the policy gap and authors a working cross-user attack; the applied fix is a *reference fix* used as a proof vehicle, not an autonomous fixer. We present this as capability evidence at small n, deliberately not as the headline.
4. **Discovery is the hard input, not a solved one** (§5). A position, not a result we measure here, well supported by prior work: access-control intent is *implicit* in code rather than stated as a spec [10], and detecting *missing* checks is specification-free and false-positive-prone even for strong tools [11]. We treat discovery as our candidate source and make no generalization claim for it.

We describe the pipeline (discover → author attack → oracle → apply reference fix → oracle re-check), then report results and an unusually explicit limitations section, because a security vendor that overclaims has already lost.

2 The authorization gap in AI-generated code

Real CVEs in AI-generated code. The Georgia Tech SSLab *Vibe Security Radar* [1] collects real CVEs introduced by AI coding tools. In separate testing of our *production static scanner*—a different system from the oracle described in this paper, used here only to characterize the problem—the pattern of *misses* is the telling signal: syntactic classes (e.g. JavaScript injection-style bugs) are caught reliably, while the residual failures cluster on authorization logic—an empty allow-list that returns "allowed," a trusted-proxy header that skips a role check—and the authorization CWEs (CWE-863 "incorrect authorization," CWE-306 "missing authentication") are exactly where pattern-based detection is weakest. Pattern matching plateaus where authorization logic begins. That is the motivation for a discovery-and-proof approach rather than more patterns; it is *not* a result of the system this paper evaluates.

An authorization CVE class, at scale. The risk is not hypothetical. CVE-2025-48757 [2] describes RLS misconfiguration and anonymous-key exposure in applications built with the AI app-builder Lovable: an analysis found 170 of 1,645 projects (10.3%) affected, with 303 vulnerable endpoints, rated CVSS 9.3 (Critical). The pattern—an AI-generated app fronting a Supabase/-Postgres database where the intended RLS policy is absent, bypassed via the service-role key, or never enforced in the route—is the exact target this paper addresses.

Why incumbents stop at the static half. Recent AI-assisted static tools have improved IDOR detection markedly. Semgrep's multimodal detector [3] reports 61% IDOR precision (versus 22% for a raw LLM baseline), 1.9× recall, and at least one IDOR found in 80% of repositories (in private beta). This is real progress on *candidate generation*—and it sharpens, rather than refutes, our thesis: once you have candidates at $\sim$61% precision, the remaining value is *proof*. A static claim that an IDOR "exists" is still a probability; a deploy decision needs certainty. VulX's contribution is the runtime oracle that converts a candidate into a proven, fixed, re-attack-survived result—or into nothing at all.

3 Threat model and scope

Target. Web applications with a clear data-ownership model, exemplified by the Next.js / Supabase / Postgres stack: framework routes or Supabase Edge Functions that read and write rows in tables whose ownership is declared by RLS policies and ownership columns (`owner_id = auth.uid()`).

Adversary. An authenticated but unprivileged user ("attacker") who can address objects they do not own—by manipulating an object identifier in a path or query parameter—and attempts to read or modify another user's ("victim") data. This is Broken Object-Level Authorization (BOLA / IDOR; CWE-639, and the related CWE-285/862/863)—the top-ranked risk in the OWASP API Security Top 10 [9].

What we prove vs. what we assume. The system *assumes* a discovered candidate names the suspect route, the owned resource, and the object-identifier source. The system *proves*, at runtime against a live instance, whether a cross-user read or write actually occurs, whether a proposed patch eliminates it, and whether the patch leaves legitimate same-user access working. The oracle's deterministic guarantees (§6) cover time-based and boolean-blind SQLi, reflected XSS, command injection, LFI/path-traversal, and read/write BOLA; *all end-to-end evidence reported here is BOLA* (§10). Stored/DOM XSS, second-order SQLi, and SSRF are explicitly out of the oracle's scope.

4 System overview

VulX is a pipeline of five stages with a strict separation of duties: LLMs propose, a deterministic oracle disposes.

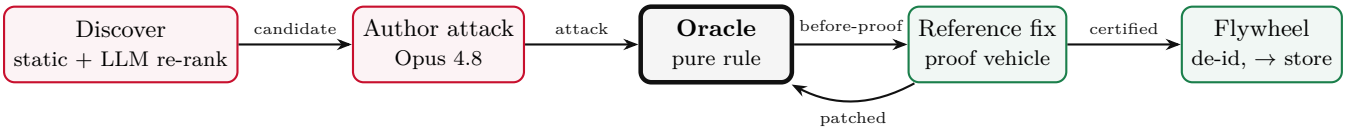

Figure 1: The VulX pipeline. Red = LLM (proposes, may hallucinate); green = deterministic. The **oracle** (black, the contribution) is invoked twice: once on the authored attack (must fire), once after the reference fix is applied (must not fire, control must pass). It alone issues a verdict; the LLM never does.

1. **Discover** (§5) reconstructs the declared authorization model from SQL migrations and routes and emits *candidates* with a confidence score. Deterministic; an optional LLM layer only re-ranks.
2. **Hacker** (§7) authors a concrete two-principal attack (attacker tries to read/write the victim's object). The model emits a *plan*; the leak/control signals are filled from ground truth, so the signal is never invented by the model.
3. **Oracle** (§6) measures exploitability before and after a patch plus a negative control, and returns one of five verdicts via a pure rule.
4. **Reference fix** (§7) is applied as the proof vehicle—the change whose effect the oracle measures. On controlled fixtures an LLM authors the candidate patch (passing a backdoor-safety gate); on real apps the fix is a known-good reference. An autonomous production fixer is roadmap (§11), so we never claim the fix step is autonomous.
5. **Flywheel** (§9) de-identifies an oracle-confirmed finding into a code-free signature for cross-codebase reuse. The de-identification gate contains no LLM.

5 Discovery: the hard problem

Discovery localizes a candidate BOLA without executing anything. Its design principle is recall-first: "a missed BOLA is worse than a false positive the oracle removes." It therefore emits *candidates, never verdicts.*

5.1 Reconstructing the declared policy

Discovery unions three deterministic extractors:

- **Route / sink loci** via structural matching (ast-grep) over Next.js App-Router handlers, Server Actions, and Supabase Deno Edge Functions, plus a body-scoped scan for untrusted inputs, auth guards, and the auth subject.
- **Data sinks and client kind**: every `supabase.from(...)` / `.rpc(...)` chain, with resolution of whether the call runs as `anon`, `authenticated`, or—critically—`service-role` (which *bypasses RLS*), by following import aliases and `Deno.env.get("...SERVICE_ROLE_KEY")` bindings.
- **RLS / ownership model**: a purpose-built Postgres parser that extracts RLS policies (including quoted policy names with spaces, which naive parsers drop), ownership columns, and predicates such as `owner_id = auth.uid()`, mapping `auth.uid()` to the JWT `sub` claim.

A candidate is raised when a handler reaches an attacker-controlled object id, queries an *owned* table, is *not* scoped by the ownership column, RLS cannot save it (service-role bypass, or an unknown client treated fail-open), and no post-fetch ownership comparison re-imposes scope. Confidence is a transparent additive prior (e.g. $+0.25$ if ownership is RLS-declared, $+0.10$ if RLS is bypassed, -0.10 if the client kind is unknown), clamped to $[0.05, 0.99]$.

5.2 The LLM is a re-ranker, not an oracle

An optional inference layer asks a frontier model whether a route's enforcement matches the declared access model. By construction it (i) never drops a candidate (recall-first), (ii) only refines confidence and reasoning, and (iii) fails soft on any model error. The unit handed downstream is a "context dossier" (intended access, missing checks, exploit scaffolding, evidence)—not a snippet and a line number. The principle is stated in the code: "an oracle proves verdicts later, an LLM's say-so does not."

5.3 What is measured

On the one *real, unmodified* open-source target (`sharespace-connect`, a Lovable-built document app on a Vite/React + Supabase Edge-Function stack), discovery localizes the BOLA at confidence **0.99** with *zero* LLM calls [MEASURED]: it walks the Deno edge function, resolves the `SUPABASE_SERVICE_ROLE_KEY` binding (RLS bypass), parses the quoted RLS policy "*Users can view own documents*" keyed to the JWT `sub`, and identifies that the route checks only that an auth header is *present*—never that the bearer *owns* the document—*deterministically, with no model.* The optional LLM inference layer (the re-ranker, §5) has been exercised against a real model only in the autonomous runs—the crafted `bola-notes` fixture and the `sharespace` autonomous run—at roughly \$0.06–0.07 each; it refines confidence but cannot fabricate a candidate. We separate the two questions discovery raises—can it *recall* a known bug shape, and can it find one end-to-end—and report them apart.

A measured recall floor (deterministic, LLM-free) [MEASURED]. We measured the *recall* of the deterministic discovery floor—the LLM-free structural pass—by running it over each app's source and asking whether it flags the oracle-proven vulnerable shape. On the six oracle-proven real-app HITs it recalls **5/6 at high confidence** (≥ 0.7) and **6/6 at a loose threshold** (the sixth, App E, only as a weak 0.45 incidental hit); on synthetic fixtures, **2/2**. We report the real

and synthetic figures separately and never blend them. An independent overfit audit confirms the floor *generalizes*: it flagged never-seen tables and skipped a benign public-data lookup purely from structural signals (RLS state, anonymous grants, PII/ownership), with no LLM in the loop.

What this is—and is not. This is *recall*, by design (discovery is recall-first); we make *no* precision claim. The floor is deliberately noisy on positives—App D's true hit, for instance, sits among 139 candidates—and the oracle, not discovery, is what removes that noise (the ≤ 5 noise cap applies only to negatives). It also measures recall of the proven shapes *from source*: the external HITs were still oracle-proven on *hand-fed* shapes, and *end-to-end* autonomous discovery beyond `sharespace-connect` (the one app auto-found end-to-end) remains the open follow-up (§14). Discovery is the *input* to this paper's contribution; the contribution we defend is the deterministic proof of a fix (§6, §10), which holds regardless of where a candidate originates. Comparative discovery against baselines (Sun et al. [10], Near and Jackson [11]) is future work.

6 The deterministic exploit oracle

The oracle is the heart of the system and the reason it is safe to run against real code. It never asks a model whether a fix worked; it *measures*.

6.1 Why "verified" usually lies

A common shortcut—one VulX's own production scanner used and which we now label "heuristically verified"—is to re-test a patched endpoint and grep the response for an error string. This passes blind/time-based injections (which leak nothing into the body), passes routes that "got fixed" by crashing (a 500 has no error string), and, in the static case, can mark a finding fixed with *zero* execution. A substring scan is not a proof.

6.2 Three measurements, one pure rule

The oracle runs a probe three times and decides with a deterministic rule:

- **Before**: fire the exploit at the unpatched app. The signal must be *present* (else the finding was a false positive).
- **After**: apply the patch, fire the *identical* probe. The signal must be *gone*.
- **Control** (negative control): a benign, legitimate request must *still succeed*. This catches "fixed it by breaking the app."

$$\mathsf{BLOCKED} \iff \underbrace{\text{exploitable\_before}}_{\text{real}} \wedge \underbrace{\neg\,\text{exploitable\_after}}_{\text{killed}} \wedge \underbrace{\text{control\_ok\_after}}_{\text{still works}} \tag{1}$$

The decision function is pure and total over five verdicts; only BLOCKED ships.

```
def decide(before, after, control_ok_after) -> Verdict:
    if before.error is not None or after.error is not None:
        return Verdict.INCONCLUSIVE      # couldn't measure -> NO PR
    if not before.present:
        return Verdict.NOT_REPRODUCIBLE  # false positive    -> NO PR
    if after.present:
```

```
        return Verdict.STILL_EXPLOITABLE # fix failed          -> NO PR
    if not control_ok_after:
        return Verdict.FIX_BROKE_APP     # broke the app       -> NO PR
    return Verdict.BLOCKED               # proven & non-breaking -> SHIP
```

Listing 1: The ship-gate. Pure, deterministic, never an LLM call.

A hardened wrapper can only ever *downgrade* a would-be BLOCKED (e.g. to FIX_BROKE_APP if the post-patch app is unhealthy, or INCONCLUSIVE if the absence is unstable across repeated measurement); it never upgrades. **Four of five verdicts open no pull request.** "No PR is safer than a false proven PR."

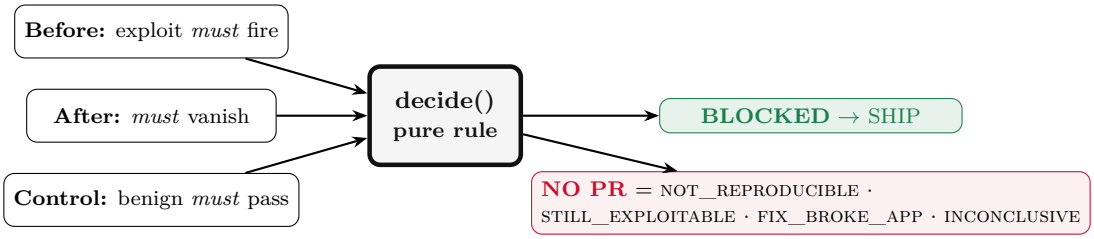

Figure 2: The ship-gate as a precision gate. Three measurements feed one pure rule. Exactly one outcome opens a pull request; the other four—including NOT_REPRODUCIBLE, which fires when a real-looking exploit does not actually occur—open none.

6.3 The precision gate: refusing what cannot be proven

The most important verdict is not BLOCKED but NOT_REPRODUCIBLE. When discovery and the LLM produce a confident, plausible candidate but the exploit *does not fire* against the live app, `before.present` is false and the oracle returns NOT_REPRODUCIBLE—**the finding is refused, not reported.** This is where an LLM-only pipeline fails: it surfaces the plausible-but-false bug as real, whereas the oracle cannot, because the verdict is a measurement, not an opinion. We show this on a real application in §8 (App F). A tool that refuses an unexploitable finding is the entire value proposition.

6.4 The BOLA signal: a leaked secret, not an HTTP 200

For read-BOLA, the signal is whether the *victim's* secret marker (a value seeded into the victim's row, never present in the attacker's request) appears in the *attacker's* response—regardless of HTTP status. This is deliberate: gating on `2xx` is both unnecessary and unsafe, because a "fix" that returns 403 *but still leaks the body* must not read as fixed. Because the marker originates in the victim's data, there is no input-reflection false positive.

For write-BOLA, the signal is a ground-truth *read-back* as the owner: did the attacker's tampering land in the victim's row (`readback_contains`), or did the attacker's delete remove it (`readback_absent`)? The write's own HTTP status is evidence only.

6.5 The negative control is what makes it sound

The control for read-BOLA is *owner-reads-own*: after the patch, the owner must still get `2xx` plus their own marker. This is the single distinguishing test between a real fix and a deny-all "fix" that locks everyone out—the latter fails the control and is correctly ruled FIX_BROKE_APP. The

control for write-BOLA is *owner-writes-own* (write then read-back), catching both deny-all-writes and silent no-op-write "fixes." BOLA is treated as an explicitly two-principal attack, so it is routed through dedicated factories rather than the single-finding probe registry.

6.6 Resistant signals and a clean reset

The injection probes are engineered against being fooled. The timing-SQLi probe requires the induced delay to *scale* with the injected sleep (`SLEEP(2)` vs `SLEEP(6)`); a flat delay (a WAF tarpit or rate limiter) reads as *absent*, not as a false positive. The differential probe normalizes nonces (CSRF tokens, timestamps) and returns *inconclusive* rather than "absent" on a nondeterministic endpoint. The canary-reflection probe checks token-level encoding so a fully entity-encoded reflection is correctly judged safe.

Soundness of the before/after comparison depends on a clean state reset between measurements. The Docker-Compose backend snapshots the seeded database as a golden template and resets via Postgres `CREATE DATABASE ...TEMPLATE` (after terminating connections and dropping the live database), then warms the connection pool. The reset path is fail-closed and is exercised in every committed run (each records its `db_resets` count).

7 From candidate to certified fix: the loop

Around the oracle sits a thin loop—*author attack* → *prove* → *apply fix* → *re-prove*—whose brains and sandbox are injected behind Python `Protocols`, so the identical loop runs offline with fakes (tests) or with a live model and sandbox. The reasoning model is hosted **Claude Opus 4.8**; a VulX-trained model is roadmap behind the same seam (§14).

The LLM's job: reason about the policy, author the attack. Given a candidate dossier (suspect route, owned resource, object-id source, and the reconstructed intended rule), Opus 4.8 answers one question: *does the route's enforcement match the intended access policy, and if not, how is the gap exploited?* Its output is concrete and checkable—a victim object, an actor, and a rationale—e.g. verbatim: "*the handler fetches the note with the service-role (BYPASSRLS) client by id only and never compares note.user_id to claims.sub, so any authenticated user can read another user's note.*" Crucially the model emits a *plan*; the leak and control signals are filled from ground truth, so the model can never fabricate the proof (§6). This is supporting capability evidence (§8), not a benchmarked rate.

The fix is a proof vehicle, not an autonomous fixer. What the oracle measures is the *effect* of a fix. On real applications the fix is a known-good *reference* patch—a vehicle to exercise the before/after/control proof—*not* the output of an autonomous fixer. On controlled fixtures we additionally let the LLM author the candidate patch (passing a backdoor-safety gate: no removed auth checks, no new `eval`/`exec`/shell primitives) and run the adversarial battle below. A productized autonomous fixer is roadmap (§11); we never claim the fix step is autonomous on real apps.

The adversarial battle (fixtures). On fixtures, once a fix holds against the first exploit, the LLM is shown the *applied patch* and asked for a bypass, up to 5 rounds, and is required to *give up honestly*—"a fix that truly holds should make you give up"—rather than invent a bogus payload. A landing bypass forces another fix, after which the oracle re-proves the new bypass *and* every prior probe. A round counts as "survived" only when a *confirmed* bypass was attempted and blocked,

so "survived N rounds" is always literally true. A run from a simulated sandbox is permanently barred from emitting a "proven" result—the real sandbox does not carry the "simulated" marker, and its absence is load-bearing.

8 The oracle in action: confirm, then refuse

We show the oracle doing the two things that matter: *confirming* real exploits, and *refusing* real-looking ones. Every figure is from a committed run-log (`simulated=false`; model `claude-opus-4-8`); apps were stood up in isolated, harness-provisioned test-beds (§12).

8.1 Confirm — a corpus of proven authorization bugs on real apps

We walk through one HIT in full, then summarize five more. **Honest scope:** of these six real-app HITs, only `sharespace-connect` was found by autonomous discovery; the other five were proven on *hand-fed* exploit shapes. They therefore strengthen the *proof* (and the range of flaw shapes the oracle handles), *not* a discovery rate (§5).

`sharespace-connect` is a real Lovable-built document app on a Supabase/Deno-edge stack. Discovery auto-found the candidate *from source* at confidence 0.97 in the autonomous run (no hand-feeding; the zero-LLM static localizer alone scores 0.99, §5); Opus 4.8 then reasoned, verbatim, that the edge function "*selects the document solely by id with .single() and never compares the row's owner_id to the authenticated caller (auth.uid()) nor checks is_public … so any authenticated user can read another user's private document.*" The oracle fired the authored attack: the attacker's response leaked the victim canary `CANARY-SHARESPACE-3df9a1c7e2` at HTTP `200` (*before*: present). A reference fix was applied; the identical probe then returned the canary *absent* at `404` (*after*: gone), and the negative control—the owner reading their own document—still succeeded. Verdict: **BLOCKED**; 3 LLM calls (cost in §10). Discovery and attack-authoring were autonomous; the test-bed was harness-provisioned; the fix was a reference patch.

Five more, across distinct flaw shapes. Batch-2 added five more oracle-proven HITs on real deployed apps—each measured before/after/control on a warm-E2B test-bed, byte-identical to its as-built source—spanning five different ways authorization breaks:

App E additionally produced an *identical* verdict on local Docker and on E2B—a backend-independent proof point: the verdict tracks the measured exploit, not the sandbox it ran in.

8.2 Refuse — the honesty contract, three ways

The system declines to confirm in three distinct ways, each a different reason a finding cannot be stood behind. (Third-party apps are anonymized; §12.)

Mode 1 — not_reproducible: a real flaw whose impact does not reproduce (App F). This is the result we most want to show, and it is fully *deterministic*: the run made **zero LLM calls**. App F was a *hand-fed oracle test*—a focused precision-gate demonstration, not an autonomous discovery run—and it contains a *genuine* authorization flaw: a sharing-grant table whose INSERT policy is `WITH CHECK (auth.uid() = <granter>)` and never checks that the caller owns the shared record, so any user can self-grant a share on another user's record. The exploit is a two-step chain—a self-grant write, then a cross-actor read of the victim's record.

The write layer behaves exactly like a real bug: the attacker's self-grant returns `201` before the fix and `403` after, while a legitimate owner-initiated share still returns `201` (flaw real; fix rejects the

App	Flaw shape	Proof (before → after; control)
App A	unauthenticated broadcast-read: an admin endpoint returns all users' rows with no auth	all-users data leaked → `401` after an auth check; control holds
App B	share-token enumeration exposing private records	anon enumerates share tokens and reads others' records → token secrecy restored; legitimate-user control holds
App C	anon table-read via a permissive RLS policy (`USING(true)`)	anon reads all private rows → policy dropped, read returns `[]`; owner control holds
App D	anon read via a view over `auth.users` exposing user PII	the view exposes PII to anon → `REVOKE` → `401`; owner control holds
App E	edge-read-BOLA: a service-role edge function reads by a caller-supplied identifier with no owner check (`verify_jwt` satisfied by the anon key)	any record's PII read by identifier (CWE-639) → `401` after an owner check; owner control `200`

Table 1: Five further real-app HITs (anonymized; §12), all BLOCKED: a real leak before, gone after a reference fix, with a passing legitimate-user control. Hand-fed shapes (not autonomous discovery); they broaden the proof across flaw types, not a discovery rate.

attacker; control holds). *Yet the oracle returns* **NOT_REPRODUCIBLE**. After the `201` self-grant, the attacker's cross-actor read returns `200` but an empty body `[]`—the planted canary never appears. The cause is a latent platform footgun: the share-view SELECT policy resolves the caller's email through a subquery on `auth.users`, but that table is RLS-enabled with zero policies, so the subquery yields `NULL`, the `EXISTS` never matches, and the read comes back empty (the idiomatic fix would key on `auth.jwt()->>'email'`).

The authorization flaw is real; the confidentiality impact is not. A scanner that flagged the missing ownership check would report a false positive. The oracle—which requires a *measured* leak, not a plausible one—declines to confirm, with no model in the loop. It is not saying "no bug"; it is saying "this real flaw does not yield the exploitable confidentiality breach we require to confirm." That distinction is the precision gate (§6) at its most valuable.

Mode 2 — inconclusive: no baseline to form a control (App G). App G leaks real data—but it is a *missing-authentication* flaw (CWE-306), not object-level authorization (CWE-639). The app exposes everything to anonymous callers, so there is no authenticated owner-read baseline against which a negative control could be defined. Rather than fabricate one, the oracle returns **INCONCLUSIVE** and refuses to issue BLOCKED—even though the leak is real. Declining to invent a control it cannot measure is the same discipline as declining to confirm an unmeasured leak, and it cleanly separates CWE-306 (missing auth) from the CWE-639 (BOLA) the oracle is built to prove.

Mode 3 — standup-blocked: the target cannot be built from source. Three targets (App H–App J) keep core tables in the Supabase/Lovable dashboard, not the repository; their RLS policies reference tables the repo never `CREATE`s. The harness cannot stand up a faithful instance without fabricating schema, so a cheap schema-completeness gate detects the gap and *defers* rather

than guess. This is not an oracle verdict but a provisioning limit—the off-repo-configuration ceiling, which turns out to be the dominant real-world wall (§11).

Together these are honest refusals in three modes—unproven impact (App F), no measurable control (App G), and an un-buildable target (App H–App J)—each chosen over reporting a finding the system cannot prove.

8.3 The full loop, on a controlled fixture (bola-notes)

On controlled fixtures we exercise the entire loop, including the LLM-authored fix and the adversarial battle. On `bola-notes` (route `web/app/api/notes/[id]/route.ts`, a service-role read by id), Opus 4.8 authored the attack ((*before*) leaked `CANARY-BOB-SECRET-7f3a9c` at `200`), then proposed a minimal, framework-idiomatic patch that passed the safety gate:

```
    const { data: note, error } = await supabaseService
      .from("notes")
      .select("*")
      .eq("id", params.id)
+     .eq("user_id", claims.sub)
      .single();
```

Listing 2: LLM-authored patch (verbatim, fixture). The single added line re-imposes the ownership check RLS would have enforced.

The oracle re-fired the probe (*after*: canary absent at `404`), the owner-reads-own control still passed, and across 8 database resets the verdict was **BLOCKED**. Shown its own patch, Opus 4.8 then tried to bypass it and *conceded honestly*—"*... the symmetric bob→alice access ... is equally blocked ... so the broken authorization is genuinely closed*"—rather than invent a payload, making the round a genuine "survived." Total LLM cost $0.084. (This is where the LLM-authored fix and battle are exercised; on the real apps above, the fix is a reference patch.)

9 The data flywheel

Each oracle-confirmed finding is distilled into a *code-free* signature so that one proven bug can drive variant analysis across other codebases. The privacy posture is the trust gate: *metadata, never code.*

The de-identification gate is a **type-total positive allowlist**—it walks the entire record and *raises* on anything not provably safe, rather than returning a partially-clean record. Keys must be known structural names; string values must be controlled categorical tokens, allowlisted CWEs, or fixed structural shapes; integers are digit-capped and floats are range/precision capped (closing a float-mantissa side channel); subclasses, bytes, and sets are refused. Defence-in-depth detectors reject emails, UUIDs, JWTs, PEM/API-key shapes, and long digit/hex runs over normalized text. **No LLM is in the gate**—a model there would itself be an exfiltration surface.

Concretely, a route like `/api/tickets/[id]` becomes the pattern `/SEG/OWNED_RESOURCE/[dyn]`; the resource becomes `OWNED_RESOURCE`; the canary value is recorded only as the category `marker`; and the cross-repo join key is a role-abstracted *authorization-pattern signature* carrying no customer noun, e.g.

```
bola|object|missing|src:route_param|sink:select:OWNED_RESOURCE_READ|
check:OWNERSHIP_PREDICATE(OWNER(RESOURCE)==ACTOR_ID)|rls_bypass|
fw:next-app-router|db:supabase-postgres
```

A build-time self-check enumerates every raw identifier from the source run as a "declared secret" and asserts none survive into the serialized payload; a ship script independently re-scans and refuses to ship on any leak. The gate was adversarially red-teamed across four passes (24 leaks found → fixed → converged to 0). The gate has now had its *first real multi-finding fill*: the four batch-2 HITs were de-identified into four distinct, idempotent, zero-raw-identifier records—the first time the flywheel was loaded with measured findings rather than modeled ones, exercising the additive, closed gate end-to-end. The remaining step is the cross-machine live transmission to the central store, which we have not yet executed; today's round-trip runs against an in-process replica of the server (§11).

10 Evaluation

This paper's contribution is the oracle, so the evaluation asks one question: *does the oracle issue correct, mechanically-grounded verdicts—confirming real exploits and, just as importantly, refusing non-reproducible ones?* We report (A) the committed end-to-end oracle runs and (B) cost.

We deliberately do *not* report a detection-rate benchmark. At this stage n is far too small to support a rate, and a headline "we find $X\%$" would invert the paper's thesis, which is about *proof*, not volume. (A separate production SAST/DAST scanner has its own benchmark numbers; we exclude them here on purpose—it is a different system, and presenting its detection rates beside the oracle would conflate two tools and overstate what *this* system has shown.)

10.1 The committed oracle runs [MEASURED]

Every row below is a *verdict*, not a flag—a committed run-log with `"simulated": false` and (where a model is used) `model: claude-opus-4-8`. The wider real-application corpus—**six confirmed HITs and five declined (two refused by the oracle, three deferred as un-buildable) across eleven real apps**—is detailed in §8 (Table 1); this section's table zooms in on the proving-ground detail: the two controlled fixtures (`bola-notes`, `bola-helpdesk`) plus the autonomously-discovered `sharespace-connect` HIT and the App F refusal. The corpus is small on purpose: a verdict that survives independent cross-checking—and a gate that *refuses* a real-looking finding—is worth more than a large table of unverified candidates, and inflating the count would contradict the discipline the oracle exists to enforce.

Three results carry disproportionate weight, and two of them are *refusals*. First, the **precision gate on a real app** (App F, §8): a *genuine* write-authz flaw the oracle declined as NOT_REPRODUCIBLE—the unauthorized write succeeded but the data leak did not—refused deterministically with zero LLM calls, where a scanner would report a false positive. Second, the **deny-all counterfactual** (`m3_denyall`): a fix that stops the leak by locking everyone out is ruled FIX_BROKE_APP because the negative control fails—soundness, not just the happy path. Third, the **real-app confirmation** (`sharespace_*`): the same bug was proven both hand-fed and via the fully autonomous discovery-plus-attack pipeline, and cross-checked from four independent "lenses," including a live deny-all disproof and a `diff` confirming the application code is unchanged by the run (we change only platform configuration to provision a test-bed; the app is left as-built and still vulnerable afterward).

10.2 Baseline: do standard static tools catch these? [MEASURED]

We ran two industry-standard static analyzers, with their standard security suites, over five real-app targets (the two oracle-adjudicated apps above plus three further candidates pending the runner):

Run	Target	Class	Verdict	Signal (before → after / control)
`m3_read`	bola-notes	read-BOLA	BLOCKED	victim canary @200 → absent @404; control ok
`m3_write`	bola-notes	write-BOLA	BLOCKED	attacker marker in owner read-back → absent; control ok
`m3_denyall` *(counter-factual)*	bola-notes	read-BOLA	FIX_BROKE_APP	leak gone but *control fails* (deny-all)
`m4_loop_again`	bola-notes	read+write	BLOCKED, then STILL_EXPLOITABLE (sibling), then BLOCKED	incomplete fix; loop converges over 2 rounds
`m5_autonomous`	bola-notes	read-BOLA	BLOCKED	Discover auto-finds (conf 0.98); 4 LLM calls
`track1_helpdesk`	bola-helpdesk	read-BOLA	BLOCKED	tenant canary @200 → absent @404; 3 LLM calls
`track1_helpdesk_run2`	bola-helpdesk	read-BOLA	BLOCKED	reproduction; identical result
`sharespace_handfed`	sharespace-connect *(our app)*	read-BOLA	BLOCKED	doc canary @200 → absent @404; control ok
`sharespace_autonomous`	sharespace-connect *(our app)*	read-BOLA	BLOCKED	Discover auto-finds (conf 0.97); 3 LLM calls
App F *(hand-fed)*	third-party *(anon.)*	write-BOLA	NOT_REPRODUCIBLE	self-grant write 201 (flaw real), fix 403, owner-share 201; *but* read-back empty—canary absent → **refused**. 0 LLM calls

Table 2: Committed end-to-end oracle runs (all `simulated=false`). The two rows in red are the point: FIX_BROKE_APP (a leak-removing fix that breaks legitimate access, correctly rejected) and NOT_REPRODUCIBLE (a *genuine* write-authz flaw whose unauthorized write succeeds but whose required data leak never materializes, so the oracle refuses to confirm a confidentiality breach). A tool that declines an unexploitable finding is the pitch.

Semgrep (project rules plus `p/typescript`, `p/javascript`, `p/security-audit`; 311 rules) and **CodeQL** 2.25.6 `security-extended` (104 queries). The vulnerable files were confirmed in analysis scope, so a miss is a query/capability gap, not a coverage gap.

Neither tool flagged a single authorization bug. On `sharespace-connect`—the one target with an oracle-*proven* BOLA—both returned zero, which we independently re-verified: standard SAST missed the exact bug VulX proved. Two honest qualifications. First, App F is invisible to both tools *by construction*: the flaw is a declarative Supabase RLS policy in a `.sql` migration, CodeQL's JavaScript extractor does not analyze SQL, and the Semgrep suites carry no Supabase-policy rules—so this is not a head-to-head detection win (and recall the oracle itself *refused* App F, §8). Second, the remaining three targets are *candidates pending* oracle adjudication by our batch runner, so their zero baseline is recorded, not yet a VulX result. **Fairness caveats:** free/standard suites only (no Supabase-specific or paid rules), a single run each, and CodeQL's JS extractor does not reach SQL. The takeaway is narrow but real: on the one proven bug in this first set—and, in Table 4 below, on all five proven HITs—two mature static tools find nothing; runtime proof is *complementary* to pattern/dataflow SAST, not redundant with it.

The same baseline on the five proven HITs. We then ran both tools over the five batch-2 HIT targets—each an oracle-*proven* BOLA (§8). The result is again **0/5**: neither tool flags any

Target (real app)	Semgrep 311 rules	CodeQL 104 queries	VulX oracle
sharespace-connect	0	0	BLOCKED (proven)
App F	0	0	NOT_REPRODUCIBLE
App K	0	1†	candidate (pending)
2 further apps	0	0	candidate (pending)
authz bugs flagged	**0**	**0**	—

Table 3: Standard static analyzers vs. the VulX oracle on five real apps. †CodeQL's single finding is `js/insecure-randomness` in a test-data helper—not an authorization issue. Neither tool flagged any authorization bug; on `sharespace-connect` (the one oracle-proven BOLA) both returned zero, independently re-verified.

authorization bug.

Target (proven HIT)	Semgrep 90 rules	CodeQL 102 queries	VulX oracle
App A	0	0	BLOCKED (proven)
App B	0	2†	BLOCKED (proven)
App C	0	—‡	BLOCKED (proven)
App D	0	0	BLOCKED (proven)
App E	0	0	BLOCKED (proven)
authz bugs flagged	**0**	**0**	—

Table 4: Standard static analyzers vs. the VulX oracle on the five batch-2 HIT targets (anonymized, App A–App E; §12), each oracle-proven. Both tools flag zero authorization bugs. †CodeQL's only findings are 2× `js/stack-trace-exposure` (CWE-209) on App B—error-handling, not authorization. ‡App C is a pure-SQL RLS bug; CodeQL's JavaScript extractor does not analyze it.

Extraction was real (e.g. App D 5,988 LoC, App A 433, App B 195, App E 78) and all files were in scope, so these are capability gaps, not coverage gaps; re-running Semgrep with `p/owasp-top-ten` added (677 rules loaded) still yielded 0/5. The reason is structural: four of the five bugs live in the Postgres/RLS layer neither tool models (CodeQL has no SQL analysis; Semgrep ships no Supabase-policy rules), and the fifth (App E) is a pure-TypeScript authorization-logic flaw outside both tools' modeled vulnerability classes. As before, this is not a head-to-head score—it shows the class sits structurally outside pattern/dataflow SAST.

Methodology note. This table counts the named community packs only (`p/security-audit`, `p/typescript`, `p/javascript`; 90 applicable rules per TypeScript target, 3 on the pure-SQL App C); we exclude VulX's own project rules from a standard-tool baseline (Table 3 included them in its Semgrep count—removing them, as here, changes nothing: still 0). CodeQL `security-extended` resolved to 102 queries here versus 104 in Table 3 (same 2.25.6, a trivial suite-resolution difference). Full method and reproduce commands are in `baseline_sast_eval.md` (§A).

10.3 Cost [MEASURED] (measured) / [ROADMAP] (modeled)

Measured cost is **$0.065–0.11 per solve** across runs (dominated by model tokens; sandbox compute is comparatively negligible). In the batch-2 corpus, cost was ∼**$0.09–0.10 per target**—e.g. the autonomous `sharespace-connect` solve at **$0.0996** on a warm-E2B template. The batch runner amortizes the sandbox: a warmed template plus parallel fan-out makes N targets cost roughly one boot (§11). For context only, MAPTA [5] reports 83% on the XBOW broken-authorization set at an average $3.67 per *repository* assessment; our per-*solve* figure is a different unit and scope, so we offer it as an order-of-magnitude data point, not a head-to-head comparison. Broader per-scan and pricing economics are *modeled, not measured*: token cost dominates (∼85–96%), and the deterministic oracle avoids paying a model to render a verdict—a structural cost edge, since the verdict is computation, not generation.

11 Limitations and honesty

We state these plainly because the product's entire premise is not overclaiming.

- **Small n, early-stage, and hand-fed.** End-to-end evidence covers eleven real apps (six *confirmed* BLOCKED, two refused by the oracle, three deferred as un-buildable) plus two controlled fixtures. Crucially, only `sharespace-connect` was found by autonomous discovery; the other real-app HITs were proven on *hand-fed* exploit shapes. We measure a deterministic discovery *recall* floor (5/6 high-confidence on the real HITs, 2/2 synthetic; §5), but make *no* precision claim (recall-first; the oracle removes the noise) and report *no* end-to-end autonomous discovery beyond the one app—both await a labeled corpus (§14).
- **The fix is a reference patch on real apps.** The before/after/control proof measures the effect of a known-good reference fix; an autonomous production fixer is *not* claimed and is roadmap. The LLM-authored fix and the adversarial battle are exercised on controlled fixtures only.
- **"Real model" ≠ trained model.** Every model-driven result uses *hosted Claude Opus 4.8*. No VulX-trained or distilled model has been exercised anywhere; the training-data pipeline exists but the model is roadmap (§14).
- **Model-guided, not unscaffolded.** The LLM emits an attack *plan* (target object, actor, rationale); the concrete leak and control *signals* are filled from candidate ground truth, and the test-bed (accounts, canary, credentials) is harness-provisioned. This is a deliberate soundness choice—the model cannot fabricate the proof—but it means we do *not* claim the LLM autonomously compromises a host with no scaffolding.
- **Autonomy boundary.** On the real OSS app the *finding* is autonomous, but the runtime test-bed (victim/attacker accounts, canary, credentials) is harness-provisioned, not autonomously discovered. The autonomous discovery path's own LLM re-ranker has been run against a real model only on a crafted fixture.
- **Off-repo configuration ceiling (the dominant real-world wall).** Many Supabase/Lovable apps keep schema and RLS in the dashboard, never in git; their RLS references tables the repo never creates, so we cannot stand up a faithful instance without fabricating schema. Three of our targets (App H–App J) were confirmed-but-unprovable for this reason—caught cheaply by a schema-completeness gate that *defers* rather than guess. In practice this is the most common

provisioning blocker, and it is the empirical case for a "connect-your-project" step over "paste-a-URL" auto-provisioning.

- **Non-authz coverage is unproven end-to-end.** The SQLi/XSS/cmdi/LFI probes are implemented and unit-tested offline, but *no* end-to-end run-log proves them against a live app. Every committed proof is BOLA.
- **Flywheel not live.** The de-identification gate is implemented and red-teamed, but the live transmission of signatures to the central store has never been executed; the round-trip uses an in-process replica.
- **Production auto-fix is heuristic.** The shipped scanner's auto-fix "verified" label is a heuristic check (substring/status, no negative control; the static branch executes nothing). The deterministic oracle described here is *not yet wired into the production scanner*.
- **Tests not re-run here.** Test counts cited from the codebase are static counts and documented claims; the suites were not executed while preparing this document.
- **Self-reported.** All results are internal and not independently audited; the run-logs are available for inspection (§A).

12 Ethics and responsible disclosure

This work studies *publicly available, open-source* applications. All exploitation was performed *solely against local clones we stood up in isolated sandboxes*—never against any third party's production system, infrastructure, or real user data. Each target ran with synthetic accounts and canary data only, and the application code was left byte-identical to its public source. One target (`sharespace-connect`, `github.com/NgoTomek`) is our own application; the others are third-party open-source projects, which we identify in this paper only in *anonymized* form.

The third-party targets are *production-grade, open-source* repositories. Several of the confirmed findings are real, not-yet-remediated authorization flaws, so **we do not disclose their identities**: every third-party app appears only in anonymized form (App A–App K), and we are in the process of privately informing the affected maintainers of the vulnerabilities we found. We report flaw *shapes*, CWE classes, and oracle results—the contribution—and deliberately omit anything that could fingerprint an app (names, verbatim policy or file names, distinctive endpoints, identifying descriptions). Going forward, external scanning operates under an explicit, scoped responsible-disclosure protocol (§14).

13 Related work

Static IDOR detection. Semgrep's multimodal detector [3] represents the current AI-assisted static frontier for IDOR (61% precision; 1.9× recall). It produces strong *candidates*; VulX is complementary in producing *proof*. CodeQL [4] pioneered multi-repository variant analysis, the model our cross-codebase flywheel adopts (§9). In our own baseline (§10.2), both Semgrep and CodeQL, run with standard security suites, flagged zero authorization bugs on our targets—consistent with the view that this class needs runtime proof, not more patterns.

Runtime proof and authorization agents. MAPTA [5], an autonomous LLM pen-tester, reports 83% on broken authorization (XBOW benchmark); we cite it for cost context (§10). BAC-Fuzz [6] argues for instrumenting the data layer rather than trusting HTTP status—the principle

behind our read-back BOLA signal. Autonomous offensive agents such as XBOW demonstrate canary/oracle assemblies; our differentiator is the *deterministic* ship-gate (never an LLM verdict), the stack-grounded policy inference that scopes the attack, and the requirement that a fix survive re-attack before anything ships.

AI-native risk. Beyond authorization, AI-generated software introduces new classes: package hallucination / "slopsquatting" (one study reports 19.7% of LLM-generated dependencies are hallucinated) [7] and retrieval poisoning (PoisonedRAG: 90% attack success with five poisoned texts) [8]. These motivate, but are out of scope for, the authorization oracle.

14 Future work

From hosted model to a trained attacker. Every oracle-confirmed run banks one labeled trajectory (schema `vulx.trajectory.v1`) whose label is the *oracle's* verdict, never a model's or a human's. Positive examples are attacker payloads that defeated a patch; negatives are bypasses the oracle blocked. This corpus is the intended training set for a distilled attacker model (SFT then RL-from-oracle), gated behind an explicit bar: a meaningful number of connected repositories *and* beating hosted Claude on held-out, oracle-confirmed exploits. Until both hold, the "trained model" remains roadmap, not product.

Cross-codebase immunity. The flywheel's end state is variant analysis: a single proven-and-fixed authorization bug, abstracted to a code-free signature, drives prove-before-surface queries across every connected codebase of the same class—isolate one vulnerability, immunize the rest. This is the most valuable idea in the system and the least built; we present it as direction, not result.

Hardening the evidence. The batch runner (warm-E2B template plus parallel fan-out), the first real multi-finding flywheel fill, a deterministic discovery *recall* floor (§5), and baselines on the proven HITs are now built and measured. The near-term priorities that remain follow from §11: discovery *precision* and *end-to-end* autonomous discovery of the new flaw shapes on a labeled corpus (the batch-2 HITs were hand-fed); end-to-end proofs for the non-authz probe families; and re-attack survival demonstrated on a real app (today it is fixture-only). An autonomous production fixer remains genuinely roadmap—real-app fixes are reference patches. The cross-machine live flywheel transmission is the last integration step.

15 Conclusion

We asked: when an LLM claims a broken-access-control bug, how do we know it is real—without trusting the model? Our answer is a deterministic oracle: a pure before/after/negative-control rule that issues every verdict, so model hallucination can never produce a false "proven." The same rule is a precision gate—on a real application it *refused* a genuine flaw whose unauthorized write succeeded but whose data leak did not (NOT_REPRODUCIBLE), exactly where an LLM-only pipeline would have cried wolf. The supporting capability is real too: Claude Opus 4.8 autonomously reasoned about a real app's access policy and authored a working cross-user exploit, which the oracle confirmed and, with a reference fix, certified closed under a passing control; on controlled fixtures the model even read its own fix and conceded honestly when it held. AI writes more code every month, and authorization is the class review was already worst at and that models are worst

at emitting. The instinct is a bigger model or more patterns; we argue the answer is *discipline*—let the model propose, and let a rule, never the model, dispose. Four of five verdicts open no pull request, and that is the point: a security tool earns trust by refusing to cry wolf. The evidence here is deliberately small and labeled, but it is *proof*, not a confidence score. That is the foundation we intend to scale.

A Reproducibility

The artifacts behind every claim in §10 are committed:

- **Run-logs.** Each row of Table 2 is a committed JSON record carrying `simulated:false`, the model id, per-step token cost, the before/after exploit signals, the negative-control result, the database-reset count, and the model's verbatim rationale.
- **The ship-gate.** The verdict is the pure function `decide()` reproduced in §6; it consumes only the three measured booleans and is independent of any model, so a third party can re-derive every verdict from the logged measurements.
- **The de-identification gate.** The flywheel signature format and its fail-closed allowlist (§9) are unit-tested, including a red-team suite that drove residual leakage to zero.
- **The baseline.** The Semgrep/CodeQL methodology, exact packs, query counts, and reproduce commands for Tables 3 and 4 are in `baseline_sast_eval.md`.

Exact run-log identifiers and the harness are available to design partners under NDA; a public artifact release will accompany the empirical version of this paper.

References

[1] Georgia Tech SSLab. *Vibe Security Radar: Real CVEs in AI-Generated Code.* `https://github.com/HQ1995/vibe-security-radar`.

[2] M. Palmer et al. *CVE-2025-48757: Row-Level Security misconfiguration in AI-generated (Lovable) Supabase applications.* 2025. Reported affected: 170/1,645 projects (10.3%), 303 endpoints, CVSS 9.3. `https://www.superblocks.com/blog/lovable-vulnerabilities`.

[3] Semgrep. *AI-Powered Detection with Semgrep (Multimodal IDOR detection).* 2026. `https://semgrep.dev/blog/2025/ai-powered-detection-with-semgrep/`.

[4] GitHub. *CodeQL and Multi-Repository Variant Analysis (MRVA).* `https://codeql.github.com/`.

[5] I. David et al. *Multi-Agent Penetration Testing AI for the Web.* arXiv:2508.20816, 2025. Reports 83% on broken authorization (XBOW benchmark), at an average $3.67 per repository assessment.

[6] *BACFuzz: Exposing the Silence on Broken Access Control Vulnerabilities in Web Applications.* arXiv:2507.15984, 2025.

[7] J. Spracklen et al. *We Have a Package for You! A Comprehensive Analysis of Package Hallucinations by Code-Generating LLMs.* In *Proc. USENIX Security Symposium*, 2025. Reported: 19.7% of generated dependencies hallucinated across 576K samples.

[8] W. Zou, R. Geng, B. Wang, and J. Jia. *PoisonedRAG: Knowledge Corruption Attacks to Retrieval-Augmented Generation of Large Language Models.* In *Proc. 34th USENIX Security Symposium*, 2025. Reported: 90% attack success with five poisoned texts.

[9] OWASP. *API Security Top 10 (2023): API1:2023 Broken Object Level Authorization.* `https://owasp.org/API-Security/editions/2023/en/0xa1-broken-object-level-authorization/`.

[10] F. Sun, L. Xu, and Z. Su. *Static Detection of Access Control Vulnerabilities in Web Applications.* In *Proc. 20th USENIX Security Symposium*, 2011.

[11] J. P. Near and D. Jackson. *Finding Security Bugs in Web Applications using a Catalog of Access Control Patterns.* In *Proc. 38th International Conference on Software Engineering (ICSE)*, 2016.