HYPOTHESIS-VERIFIED NEURAL-SYMBOLIC TAINT ANALYSIS FOR VULNERABILITY DETECTION IN PYTHON LIBRARIES

Anonymous authors
Paper under double-blind review

ABSTRACT

Static taint analysis is widely used for vulnerability detection, and one of the factors that decides its precision is the treatment of sanitizers: mainstream analyzers model a sanitizer as a fixed cleansing rule whose effectiveness follows from rule existence alone. In Python, where sanitizing behavior is conditional on runtime types, control flow, and library versions, this assumption yields false negatives and false positives. We propose SanTA, a neural-symbolic method in which the conditions for a sanitizer to take effect are no longer unilaterally assumed by the analyzer but exist as hypotheses to be verified. SanTA uses a large language model to identify candidate sanitizer calls and formulate effectiveness conditions in a closed vocabulary; a symbolic layer verifies each against source code facts, and the call blocks taint only when all conditions and obligations are established, with auditable evidence on every verdict. On the CVEfixes dataset SanTA attains the best F1 (0.67) and recall (0.83) of the analyzer roster; on our conditional-sanitizer benchmark, ablations attribute the gains to the verification layer. A campaign over 30 popular open-source Python packages surfaced 15 previously-unknown vulnerabilities, 3 with assigned CVE ids.

1 INTRODUCTION

Taint analysis tracks untrusted data from sources to security-sensitive sinks(Livshits & Lam, 2005; Newsome & Song, 2005; Wang et al., 2025a; Liu et al., 2025), and one of the factors that decide its precision is the treatment of sanitizer nodes on taint paths. Once the effectiveness of a sanitizer is misjudged, the analyzer commits a false negative or a false positive. Mainstream static taint analyzers model sanitizers as rules similar to sources and sinks, and assume sanitizer effectiveness is binary, determined only by whether a rule is present(Chiang et al., 2024; Ghebremichael et al., 2026; Rizk et al., 2026). For statically typed languages, where types, overloads, and control flow are fixed at compile time, this assumption largely holds. In Python, sanitizing behavior is conditional: it depends on runtime control flow, credential state, and library versions(Gu et al., 2026). When sanitizer effectiveness is undecidable, a static analyzer either ignores the call and yields false positives or trusts it blindly and yields false negatives. An empirical study of Python SAST tools reports that no single tool detects more than 40% of the vulnerabilities in its benchmark, and the dominant false-positive source is flagging pattern matches without verifying whether sanitization is applied (Liu et al., 2026). Sanitizer conditionality is a structural consequence of Python's dynamic semantics: as long as sanitizers are treated as unanalyzable cleansing primitives, these errors will keep recurring. Conditionals are prevalent in hand-written sanitizer functions in other languages, and control-flow-insensitive handling of this structure yields numerous false positives (Klein et al., 2022; Xie & Aiken, 2006).

Systems such as CodeQL (Semmle/QL developers, 2025) and Pysa (Meta/Pysa developers, 2025) offer extension mechanisms through which analysts can author custom sanitizer models, but writing such models requires expert knowledge that is chronically scarce, leaving real-world library code under-covered.

Insight. We therefore turn sanitizer condition of effectiveness into a hypothesis to be verified. In order to mitigate the false positives and false negatives that sanitizer conditionality causes in Python

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.

taint analysis, we use the LLM to propose what the sanitizer on a taint path does and under what conditions it works by reading code semantics, whereas the symbolic layer checks with a solver and against the real program whether those conditions hold, and contributes the verdicts. This division of labor—identification supplies, verification decides—holds across all three evaluated backends (§6.2).

We implemented this insight as a pipeline based on LLM hypotheses and symbolic verification. For each taint path that traverses a sanitizer call, SanTA (i) expands the call chain to expose the library-internal component that actually performs cleansing, (ii) asks the LLM to identify that component, name its data parameter, and state effectiveness conditions in a closed seven-kind vocabulary, (iii) encodes those conditions into Z3 through a trusted, total encoder, (iv) checks guard correspondence, whether values that violate a condition are provably rejected by a fail-closed guard inside the sanitizer or in its path callers, and (v) for two sanitizer families that conditions cannot express, verifies output-side postconditions or compiles the function into a closed rewriter vocabulary and checks it by sandboxed differential execution. Only when every condition and every verification obligation is established does the call block taint.

Contributions.

1. A neuro-symbolic taint-analysis framework in which the LLM is strictly a hypothesis generator and every verdict is issued by a mechanical layer (Z3, guard correspondence, differential execution), so that each blocking decision carries auditable source evidence.
2. An evaluation under one frozen read-out against CodeQL, Pysa, Bandit, and LLM-agent baselines: on the held-out CVEfixes pool (163 CVEs) SanTA attains the best F1 (0.67) and recall (0.83) of the roster; attribution ablations on the 50-CVE sanitizer-conditioned benchmark localize the gains to the adjudication layer (§6.3).
3. CVE-SAN-Python, a high-quality 50-CVE benchmark of conditional sanitizers, and a wild campaign on 30 popular Python packages that surfaced 15 previously-unknown vulnerabilities, 3 with assigned CVE ids.

2 Background

2.1 Prerequisites

Taint analysis decides whether untrusted data can travel from a source s to a sink t(Livshits & Lam, 2005; Newsome & Song, 2005; Wang et al., 2025a; Liu et al., 2025). If an effective sanitizer sits on the path, taint is cleared there and the path collapses. Whether the sanitizer is effective is exactly the problem. Let Π be the set of taint paths of program P. A path $\pi \in \Pi$ is a propagation sequence from s to t; a candidate call site on π is c, its callee f, its argument tuple a. The analyzed program means the target package plus the source-readable part of its dependency closure; a fact outside this scope is unavailable.

Blocking assertion. If f performs cleansing at c, then cleansing at c is a blocking assertion over two kinds of conditions: declared effectiveness conditions $\Phi(c) = \{\varphi_1, \ldots, \varphi_m\}$, and engine-side verification obligations $\Omega(c)$ built from the call site and its path context:

$$\mathrm{blocks}(c) \triangleq \bigwedge_{i=1}^{m} \varphi_i \wedge \bigwedge_{o \in \Omega(c)} o. \tag{1}$$

Only when every condition and every obligation is established by program facts does $\mathrm{blocks}(c)$ hold and taint die at c. The status of φ_i is the core of the method: it is not a constant fixed by the analyzer at analysis time but a predicate about the program that must be adjudicated on the analyzed program's own source.

2.2 Related Work

Sanitizer modeling in static taint analysis. CodeQL (Semmle/QL developers, 2025) and Pysa (Meta/Pysa developers, 2025) express sanitizers as named cleansing rules whose effectiveness follows from rule existence; extension requires analyst-written specifications. Automatic sanitizer

treatment has its own line: VulChecker identifies sanitizers through semantic analysis to sharpen taint tracking (Chen et al., 2021); Saner composes static and dynamic analysis to expose faulty sanitization procedures that attackers can bypass (Balzarotti et al., 2008); a sanitizer-centric PHP analysis locates the precision bottleneck in incomplete recognition of user-defined sanitizers (Su et al., 2022); and XSSky derives concrete malicious inputs whose survival of sanitization proves a vulnerability (Shi et al., 2025). Formal precedents are older: BEK checks sanitizer behavior in a dedicated algebra (Hooimeijer et al., 2011) and Ostrich decides string-transformer safety by SMT over transducers (Chen et al., 2022); both require fixed-DSL specifications, whereas SanTA derives call-site conditions from in-the-wild Python as LLM hypotheses. These systems establish that sanitizer status governs precision, but each treats the sanitizer as an identified whole that either executes faithfully or is probed with concrete inputs; none declares and mechanically adjudicates the conditions under which a specific call cleanses. SanTA does not replace such engines but re-models the sanitizer from a named rule into a condition-bearing hypothesis object adjudicated by program facts with recorded evidence.

LLM-assisted vulnerability detection. IRIS (Li et al., 2024) infers taint specifications with LLMs; VulAgent (Wang et al., 2025b) organizes detection as hypothesis-validating agents; LLMxCPG (Lekssays et al., 2025) uses an LLM to query a code property graph for minimal slices. In all three, model output directly or filter-wise becomes the conclusion. STaint couples bi-directional taint tracking with LLM semantic judgment to detect second-order PHP vulnerabilities (Ji et al., 2025); QLCoder synthesizes reusable CodeQL queries from CVE descriptions through an agentic pipeline (Wang et al., 2026a); and an LLM-driven recognition method lets the model issue sanitization determinations guarded by confidence thresholds and selective symbolic-execution fallback (Meng et al., 2025). SanTA restricts the model to hypotheses; conclusions come from mechanical adjudication, and unsupported judgments cannot enter results.

Neuro-symbolic verification. LLM-Sym (Wang et al., 2024) lets the LLM write solver code, whose translation fidelity is unverifiable by construction; we adopt its propose-solve-refine pattern but replace free-form solver authorship with a closed vocabulary and a trusted total encoder, preserving verifiability. The hypothesis-verification paradigm makes analyzer assumptions explicit and checkable; SanTA applies it to sanitizer conditions and separates contradiction from fact-absence in every record. Symbol-LLM (Xu et al., 2024) proposes a two-stage SFT framework that ensures a balance of capabilities between symbolic language and natural language while incorporating symbolic knowledge. Empirical evidence indicates that LLMs reason about execution paths more reliably as a complement to symbolic techniques than as their replacement (Wang et al., 2026b); GraphLedger validates repository-level assumptions against explicit evidence held in an assumption ledger (Le et al., 2026), applying the same paradigm at a coarser granularity.

3 Motivating Examples

To demonstrate the prevalence of this type of issue, we collected 50 real-world CVEs that exhibit this pattern into a benchmark dataset. Two representative examples are shown in Figure 1. In CVE-2025-61911 in python-ldap(National Institute of Standards and Technology, 2025), a string-escaping sanitizer is expected to neutralize special characters in a filter assertion. Its cleansing effect depends on two runtime conditions: the actual type of the assertion value, and the value of the escape-mode parameter. When the mode is non-default and the input type mismatches, the sanitizer neither raises nor escapes but returns the dangerous content unchanged, letting the caller assemble an injectable filter. The fix adds an unconditional type check at function entry, closing exactly this conditional path. In CVE-2026-47392(National Institute of Standards and Technology, 2026), sandbox protection is a set of whitelists: attribute names, callable names, and string-constant checks. Protection holds only while runtime-constructible values stay inside the lists; an attacker reaches a dangerous operation through an indirect path the lists do not cover. The bypass survives an earlier patch of the same kind.

The common structure: the sanitizer's effectiveness is determined by a runtime-varying value that can be neither enumerated from source nor covered by the list itself. A conditional sanitizer is a function or guarded code fragment on a taint path between source and sink whose cleansing holds only when runtime-varying conditions on its inputs are satisfied. This is not an isolated bug pattern but a class of conditional sanitizers. Existing static analysis treats them as unconditional barriers:

CVE-2025-61911: python-ldap.py escape_filter_chars

```
def escape_filter_chars(assertion_value, escape_mode=0):
    ...
    s = assertion_value  # No type check on the input
    if escape_mode == 0:
        s = s.replace(...)
    elif escape_mode == 1:
        # The escaping logic is applied only to string inputs. For non-
string inputs, this branch is silently skipped and the original value is
returned unchanged.
        ...
    elif escape_mode == 2:
        ...
    return s
```

CVE-2026-47392: python_tools.py

```
_blocked_attrs = {'__subclasses__', '__globals__', '__class__'}
_blocked_calls = {'exec', 'eval', 'compile'}

def validate(tree):
    for node in ast.walk(tree):
        if isinstance(node, ast.Attribute) and node.attr in _blocked_attrs:
            raise SecurityError(f"Blocked attribute: {node.attr}")
    # The other checks (call-name and string-constant) are omitted
here.
def execute(code):
    tree = ast.parse(code)
    validate(tree)
    exec(code, {})
```

Figure 1: **Two conditional-sanitizer vulnerabilities.** Cleansing holds only under a runtime condition (input type and escape mode; whitelist membership of runtime-constructible values), so the same call both sanitizes and fails to.

ignoring the condition causes false negatives, trusting it blindly causes false positives. Detecting this class accurately requires re-modeling the sanitizer as a conditional object and verifying, along each concrete path, whether its conditions hold. That is the problem SanTA solves.

Adjudication target. Does there exist $\pi \in \Pi$ reaching t such that every sanitizer call c before t on π satisfies $\neg\,\mathrm{blocks}(c)$? Generation is deterministic given a condition set, and no voting over samples ever decides program facts. Every decision is emitted as an adjudication record in a uniform, auditable format.

4 METHOD

We designed SanTA, which extracts conditional sanitizer paths, formulates conditional hypotheses, and issues verification decisions in three steps. The first step, presented in §4.1, expands each sanitizer candidate into its full call chain and identifies, from a semantic slice of the path, the core function that actually performs cleansing. The second step, given in §4.2, identifies the data parameter of the core function and states its effectiveness conditions as predicates in a closed seven-kind vocabulary. The third step, developed in §4.3, checks each condition against program source facts and issues the blocking verdict. Figure 2 gives an overview, and Algorithm 1 summarizes the three steps.

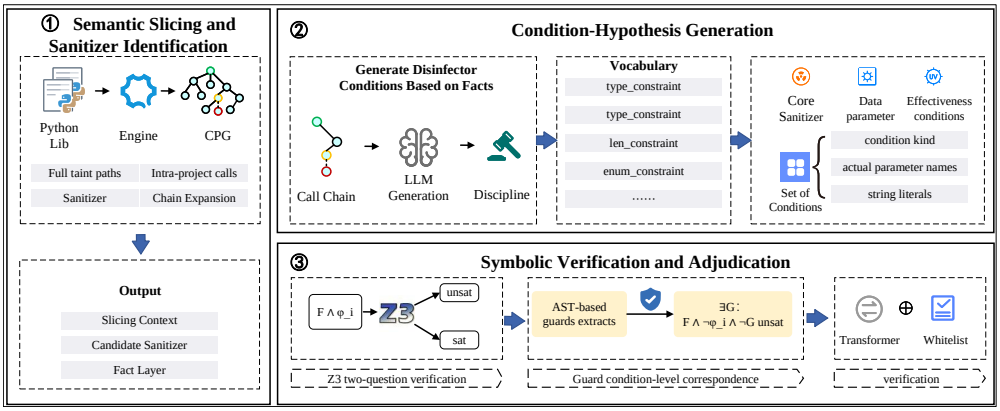


Figure 2: **SanTA overview.**

4.1 STEP 1: SEMANTIC SLICING AND SANITIZER IDENTIFICATION

This step locates the call sites on taint paths that perform cleansing, shrinking the adjudication object from whole paths to concrete call sites. The underlying engine's output is designed for adjudica-

Input: program P, taint paths Π **Output:** verdicts V, adjudication records A

$V \leftarrow \emptyset$; $A \leftarrow \emptyset$
for each path $\pi \in \Pi$ **do**
  for each candidate call site $c \in \pi$ **do**
    expand chain around c; extract call-site facts $\mathcal{F}$
    $f, p_{\text{data}}, \Phi(c) \leftarrow \text{LLM}(\text{slice}(c))$
      ▷ closed vocabulary; one refine round on mechanical error
    for each $\varphi_i \in \Phi(c)$ **do**
      $(\text{status}_i, \text{witness}_i) \leftarrow \text{Z3}(\mathcal{F}, \varphi_i)$
      if $\text{status}_i \in \{\text{violated, attacker-violable}\}$ **then**
        rescue only if Eq. equation 2 holds for some guard G
          (in the core or its path closure)
    route by family: transformer output-side proof / whitelist
      compile, differential execution, containment
    $v_c \leftarrow \bigwedge_i \varphi_i \wedge \bigwedge_{o \in \Omega(c)} o$
      ▷ Eq. equation 1; every term recorded in A
return V, A

Algorithm 1: Sanitizer identification, condition hypotheses, and adjudication.

tion completeness: (i) full call relations are preserved, so argument provenance and return use stay connected; (ii) the engine emits every rule-matched path without pre-pruning weak matches; (iii) a sanitizer-rule match is deliberately low-confidence: matching a sanitizer specification does not clear taint in propagation but marks the call site as a candidate, leaving the decision to later stages. Named rules that never enter this flow keep the engine's original behavior.

Chain expansion and the fact layer. The call chain stops at the sanitizer's entry layer; the component that actually cleans sits deeper. SanTA expands from the flow's endpoint functions, the engine call chain, and the sanitizer's definition and callers, along intra-project call edges, with a bounded AST BFS. In parallel, a no-LLM fact layer extracts static types and value sets of the sanitizer call-site arguments. These are pinned as facts in Z3. The LLM gets the slice, including source text, argument provenance and usage, and the sink call form, to analyze. It answers whether the slice contains a call performing cleansing and names the callee.

4.2 Step 2: Condition-Hypothesis Generation

This step reduces the runtime question of whether a sanitizer takes effect to predicates over program source, which the symbolic layer then verifies individually. The LLM emits three fields. The core sanitizer is the function on the call chain that actually performs the cleansing or rejection operation. The data parameter is the formal parameter of the core function that carries the attack data. The effectiveness conditions are predicates drawn from a closed vocabulary of seven kinds, summarized in Table 1.

Table 1: **Closed condition vocabulary.** Each kind has a fixed, total mapping into Z3. Anything outside the vocabulary is equivalent to not established in Eq. equation 1; the LLM cannot inject arbitrary solver semantics.

kind	meaning	Z3 encoding
type_constraint	input must have type T	$\text{ty} = T$
enum_constraint	value in enumeration V	$\bigvee_{v_i \in V} x = v_i$
range_constraint	numeric range	$\ell \le x \le u$
excludes_constraint	value contains none of s_i	$\bigwedge_i \neg \text{Contains}(x, s_i)$
prefix_constraint	value starts with some p_i	$\bigvee_i \text{PrefixOf}(p_i, x)$
regex_constraint	(non-)match of a regex	$\text{InRe}(x, \langle r \rangle)$, negatable
len_constraint	bounded length	$\ell \le \text{Length}(x) \le u$

Discipline. The model output is parsed as data under a fixed schema. Three degrees of freedom survive validation: the choice of condition kind, the actual parameter names, and the string literals; every other field is discarded on entry. Conciseness rules forbid character-by-character enumeration and cap the size of substitution lists. A single self-correction round returns mechanical errors to the model, covering output that fails to parse and a proposed core function outside the call chain. The vocabulary forms the interface between the method and the analyzed language: a condition enters the blocking assertion only if some source fact can support it.

4.3 Step 3: Symbolic Verification and Adjudication

The symbolic layer adjudicates each condition and each obligation on the analyzed program's source facts, then issues the blocking verdict (Algorithm 1, lines 6–11). It has four cooperating components.

(a) Z3 two-question verification (de Moura & Bjørner, 2008). For each condition φ_i, with the fact layer's pinned input class $\mathcal{F}$: $\mathcal{F} \wedge \varphi_i$ UNSAT $\Rightarrow$ violated, facts prove the condition cannot hold; a witness is extracted; $\mathcal{F} \wedge \neg\varphi_i$ UNSAT $\Rightarrow$ established, no value in the class satisfies the negation; otherwise attacker-violable. Queries run under Z3 4.11.2 with a 30 s timeout; UNKNOWN/timeout routes to undecided, which never blocks.

(b) Guard condition-level correspondence. Violation is not vulnerability: the function may reject violating inputs, fail-closed. SanTA extracts guards from the core function's AST, such as conditional rejections, assertions, and tail fallbacks. We encode test expressions T through a whitelist and rescue a violated condition only under condition-level correspondence:

$$\exists \text{ guard } G \text{ on the same parameter:} \quad \mathcal{F} \wedge \neg\varphi_i \wedge \neg G \ \text{ UNSAT}, \tag{2}$$

i.e., every value violating φ_i provably triggers G and is rejected. When the sanitizer core is predicate-shaped, the core itself has no guards and the fail-closed check lives in path callers; SanTA synthesizes caller guards from the taint-path closure. Restricting synthesis to the path closure is what prevents guards from unrelated flows from masquerading as evidence.

(c) Transformer output-side verification. The vocabulary expresses input preconditions, but unconditional transformers are safe for any input, because their guarantee is an output postcondition with no meaningful precondition. For single-state transformers, SanTA extracts the output declaration set and mechanically proves the universal postcondition, blocking the flow with evidence grade transform_verified.

(d) Whitelist rewriter verification. Whitelist rewriters, such as URL-scheme filters, are also outside the condition vocabulary: their safety lies in what the rewrite lets through, not in any precondition on the input. SanTA therefore has the LLM compile the function into a model written in a closed rewriter vocabulary. Here the LLM is a compiler, not a judge. The compiled model is verified by sandboxed differential execution against the real function and its module-level closure, on a 255-item adversarial corpus of dangerous URL schemes under case, control-character, and entity obfuscations, together with whitelist positives and seeded fuzz inputs. The model receives at most one self-correction round; if a mismatch remains afterwards, the model is rejected.

Conjunctive adjudication and reliability gates. The unit verdict is ALIVE or BLOCKED: one unblocked attacker path suffices for ALIVE, while BLOCKED requires every flow blocked plus at least one decisive data condition, either a content constraint (enum, range, excludes, prefix, regex, len) or a guard-rescued violation; a vacuous `type is str` does not count, since string contents stay attacker-controlled. Unencodable conditions, an unidentified core sanitizer, and undecided verification never block a flow; cleansing counts only when proven. Deny-list rewriters admit no sound BLOCK evidence and keep the conservative alarm.

5 Experimental Design

Datasets. (1) CVE-SAN-Python, built by us: 50 manually reviewed real-world CVEs in Python projects, each with complete vulnerable and fixed versions, 100 standalone units in total (construction and annotation in Appendix E). (2) CVEfixes: the full CVEfixes Python pool (Bhandari et al., 2021), 297 snapshots read out at CVE granularity as the held-out generality benchmark; the two benchmarks share no CVE (funnel and read-out in Appendix C). (3) Wild campaign: 30 analyzed releases of 30 popular open-source Python packages, none of which has a fix at analysis time.

Metrics. Unit-level precision P, recall R, F1, and false-discovery rate $\mathrm{FDR} = 1 - P$. Component metrics: core-sanitizer identification rate, condition status distribution by kind, unencodable share.

Baselines and controls. (i) Engine baseline: our in-house Python taint-analysis engine, run with its built-in sanitizer specifications cutting flows as usual—the substrate SanTA extends. (ii) Cut-on-sight: any sanitizer call on the path blocks it. (iii) CodeQL (CLI 2.22.2; the main table reports its strongest of three configurations). (iv) Pysa: declarative models built from the same extracted rule information. (v) Bandit. (vi) VulAgent and LLM-Sym, issuing unit-level verdicts. Code, prompts, rules, adjudication records, and both benchmarks will be released with the paper.

LLM configuration. A single backend, qwen3-next-80b-a3b-instruct, serves every LLM stage throughout the evaluation under one frozen decoding configuration; the stages are slice identification, condition-hypothesis generation, and rewriter compilation. Appendix B varies the backend and the fusion arm remains the best configuration on each.

6 Results

6.1 RQ1: End-to-End Detection

Table 2: **Detection on the full CVEfixes Python pool and the 50-CVE sanitizer-conditioned core of CVE-SAN-Python.** Left: 163 CVEs. Right: 100 standalone pole units. All arms run under one frozen read-out; SAST alerts are anchored to label functions; LLM arms issue unit-level verdicts.

	CVEfixes pool (CVE-level, 163 CVEs)					**CVE-SAN-Python core (unit-level, 100 units)**				
Method	P↑	R↑	F1↑	TP	FP	P↑	R↑	F1↑	TP	FP
SanTA (ours)	**0.56**	**0.83**	**0.67**	**135**	**105**	**0.53**	**1.00**	**0.69**	**50**	**45**
CodeQL	0.54	0.48	0.51	78	67	0.39	0.10	0.16	5	8
Pysa	0.38	0.02	0.04	3	5	0.00	0.00	0.00	0	3
Bandit	0.53	0.59	0.56	96	87	0.41	0.48	0.44	24	35
VulAgent	0.52	0.69	0.59	113	104	0.53	0.20	0.29	10	9
LLM-Sym	0.51	0.34	0.41	56	55	0.48	0.20	0.28	10	11

We evaluate end-to-end detection on the full CVEfixes Python pool and CVE-SAN-Python against CodeQL, Pysa, Bandit, VulAgent, and LLM-Sym. On CVEfixes, a CVE is a true positive if any of its vulnerable snapshots is flagged and a false positive if any of its fixed snapshots is flagged, so one CVE can contribute both; the 163 CVEs form the recall denominator (Appendix C). On CVE-SAN-Python we report the full core under the same frozen read-out at unit level; both are shown in Table 2. On the full pool, SanTA attains the best F1 (0.67) and recall (0.83) of the roster, against CodeQL's 0.48. On the sanitizer-conditioned core, the adjudication layer lifts F1 from 0.65 (engine baseline) to 0.69 while raising recall to 1.00, whereas cut-on-sight collapses recall to 0.48. The conclusion: SanTA converts rule information into verified detection at CVE granularity, and where sanitizers condition effectiveness, the adjudication layer, not rule coverage alone, decides the outcome.

6.2 RQ2: Component-Level Behavior

On the 71 adjudicated flows of CVE-SAN-Python's 20-CVE expert core, the ground-truth-annotated subset behind all component metrics (annotation protocol in Appendix E), the LLM identified a core sanitizer on 68 (0.958) and named the data parameter on 51 of them (0.750); of 75 declared conditions, 5 were established, 2 violated, 59 attacker-violable, and 9 unencodable. The unencodable conditions mark the vocabulary-capacity boundary, visible rather than hidden (Figure 3a). Every non-established condition carries its failure cause in the record.

Condition-sanitizer identification accuracy. To verify that each pipeline stage is individually measured, rather than trusting end-to-end F1 alone, we score localization, condition recognition, and verification separately on the expert core against per-CVE ground truth that annotates the core sanitizer, its family, and the fix-implied condition kind and data parameter. Chain expansion lifts core-sanitizer localization from 0.15 to 0.49; declared conditions match the ground-truth kind and data parameter on 0.45 of core-located input-family flows; and the mechanical verdict agrees with

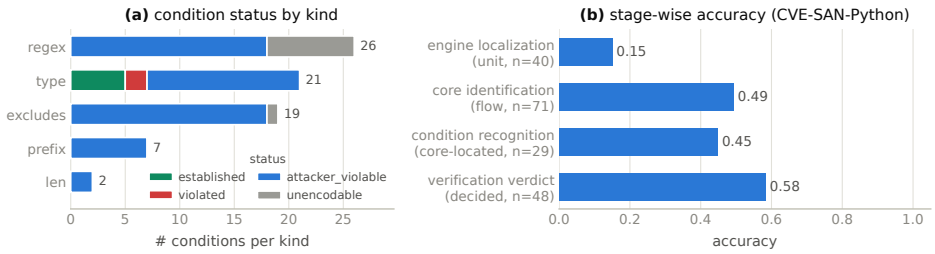


Figure 3: **RQ2 on the CVE-SAN-Python expert core: condition status by kind (a) and stage-wise identification accuracy (b).** (a) Most declared conditions are attacker-violable (blue), the honest signature of conditional sanitizers; unencodable (grey) is the vocabulary-capacity cost, routed conservatively. (b) Engine localization is unit-level (pre-expansion); core identification is flow-level after chain expansion; condition recognition is scored on core-located input-family flows, the verdict on decided flows.

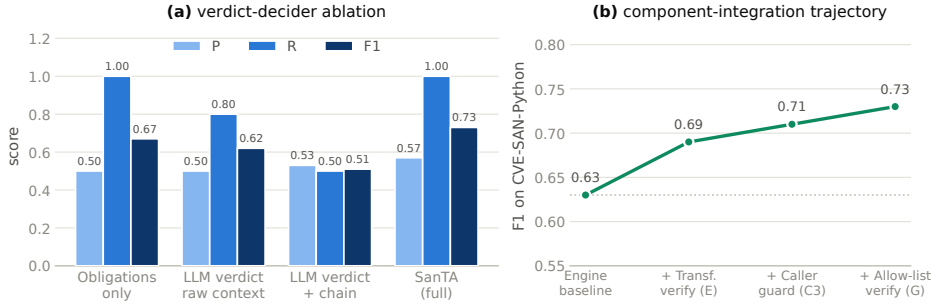


Figure 4: **RQ3 attribution on the CVE-SAN-Python expert core: verdict-decider ablation (a) and component-integration trajectory (b).** (a) Each arm removes or replaces one component of the adjudication layer; whoever issues the verdict is the only difference. (b) F1 with recall held at 1.0 throughout; E: transformer output-side verification; C3: predicate-caller guard synthesis; G: whitelist rewriter compile, differential execution, and structural containment. Each step is a soundness-bearing verification family added to the adjudication layer, not prompt tuning.

the pole expectation on 0.58 of decided flows, with a pole-asymmetric split (0.96 vulnerable vs 0.24 fixed) that reappears as the false-positive equilibrium of §6.3; undecided flows never block (Figure 3b). A backend-generalization run over the three backends of Appendix B (thinking mode enabled, protocol frozen) shows that core-sanitizer identification is backend-robust, that the pure-LLM identification-and-judgment arm never clearly improves on the engine, and that the full neuro-symbolic composition attains the best F1 on every backend (Appendix B).

6.3 RQ3: Ablation Studies

Attribution logic. The evidence is threefold, on benchmarks that exercise the layer end-to-end. First, the component-integration trajectory (Figure 4b): on the expert core every unit is adjudication-touched by construction, each rescue family added to the layer is a soundness-bearing verification, and F1 moves from 0.63 to 0.73 with recall held at 1.00; cut-on-sight on the same units collapses recall to 0.55. Second, the verdict-decider ablation on the same core (Figure 4a) pins the division of labor. The obligations-only arm establishes no block at all, so every blocking proof in the full method rides on LLM-declared conditions. Moving the verdict to the LLM collapses recall to 0.80 and 0.50, while the composition keeps recall 1.00 and converts five fixed poles to true negatives. Third, on the full CVEfixes pool (Table 2) the pole-symmetric false-positive mass persists under the adjudication layer, so the held-out contribution is verified detection without added precision loss rather than a dev-set artifact.

6.4 RQ4: Wild Discovery on Popular Python Libraries

The benchmarks are retrospective; this RQ asks whether the frozen pipeline finds real, previously-unknown vulnerabilities in the wild. We ran the pipeline over 30 analyzed releases of 30 popular open-source Python packages spanning agent and LLM frameworks and widely used libraries from web serving, data processing, crawling, and templating, each pinned to a release. Unknown bugs have no fix to consult, so no diff or patch information can enter these verdicts by construction. Confirmed findings were reported to the maintainers through huntr and VulDB; the submission ledger is frozen. The campaign surfaced 15 vulnerabilities confirmed or triaged by the receiving vendors: 12 command injections (CWE-78), 2 code injections (CWE-94), and 1 regular-expression denial of service (CWE-1333). Three received CVE ids (withheld for double-blind review), and the remainder are under maintainer triage. The other 15 analyzed releases produced no submittable finding: their execution sinks are absent, admin-gated, vendor-documented, or local-trust. The confirmed findings concentrate in one recurring pattern: a package forwards configuration-controlled values into process or code execution, and the decisive work on each is done by the adjudication layer, which is the paper's innovation over the engine substrate. The construct guarding the sink (a type or non-empty-ness check, a command-name whitelist, an http-prefix routing test) is never trusted as a cleansing rule: the model re-models it as a conditional sanitizer and declares its effectiveness conditions in the closed vocabulary, and the symbolic layer adjudicates each condition against the source facts. On every confirmed finding, the adjudication returns attacker-violable with a witness input that satisfies the guard yet still carries the attack, and the guard-correspondence obligation of Eq. equation 2 fails to rescue the violation; since no condition is established, the call cannot block taint and the alarm survives, carrying the witness and the failed obligation as its evidence record. This mechanically recorded evidence, not model opinion, is what the vendor reports rest on: an engine that trusts the guarding rule clears the path and records nothing, and an unconstrained LLM verdict asserts the bypass without proof. The public reports disclose the sanitizer-condition structure only; exploit construction stays with the maintainers.

7 Discussion

Internal validity. LLM generation is stochastic; we freeze prompts and temperature, cache responses, make adjudication deterministic given a condition set, and share the engine and read-out across arms, so metric deltas stem from the adjudication layer. CVE-SAN-Python expert-core numbers are development-set numbers (behavior discrimination, not unbiased generalization); the full CVEfixes pool supplies the no-iteration counterpart, with exact McNemar tests and bootstrap 95% intervals in Appendix D.

Scope and external validity. SanTA addresses taint-shaped vulnerabilities: flows from a recognizable source to a recognizable sink whose safety hinges on a conditional sanitizer. Vulnerability classes without this shape, from memory-safety bugs and race conditions to authentication, business-logic, and design-level flaws, have no sanitizer-condition structure to adjudicate and are beyond the method. Both benchmarks sample the Python ecosystem; transfer to other dynamic languages (e.g., JavaScript, Ruby) requires rebuilding the trusted encoder and guard extraction per language and remains unexplored. The trust computing base beyond the trusted encoder is enumerated in Appendix F; the unexamined premise—static decidability of sanitizer effectiveness—is delimited by the false-positive equilibrium of §6.3.

8 Conclusion

Misjudging sanitizer effectiveness is a structural source of false positives and false negatives in dynamic-language taint analysis. SanTA turns the analyzer's unilateral assumption into a verifiable hypothesis: an LLM declares effectiveness conditions in a closed vocabulary, a symbolic layer adjudicates each condition and each obligation on program facts, and the call blocks taint only when all are established, every decision carrying auditable evidence. On the full CVEfixes pool SanTA attains the best F1 and recall of the roster, and attribution ablations localize the gains to the adjudication layer. In the wild, the same pipeline surfaced 15 previously-unknown vulnerabilities in popular Python packages, 3 with assigned CVE ids. The usable surface is bounded by vocabulary capacity and fact availability; extending both is future work.

Reproducibility Statement. All experiments run under one frozen read-out: engine, baselines, and LLM configuration (§5); adjudication protocol and closed vocabulary (§4.3, Algorithm 1); determinism and frozen decoding (§7); datasets and ground truth (Appendices C, E); case studies and backend protocol (Appendices A, B); statistical protocol (Appendix D). The wild-campaign units and the anonymized source code are submitted as supplementary material.

AI USE STATEMENT

In this work, we used generative AI tools only for language polishing, grammar checking, and improving the presentation of figures and tables. We did not use them for automated research, theoretical analysis, experimental design, or data collection. The data used in this work come from real references and open-source platforms such as NVD. We have reviewed all AI-assisted outputs and take responsibility for the final content.

REFERENCES

Davide Balzarotti, Marco Cova, Viktoria Felmetsger, Nenad Jovanovic, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. Saner: Composing static and dynamic analysis to validate sanitization in web applications. In *2008 IEEE Symposium on Security and Privacy (S&P 2008)*, pp. 387–401, 2008.

Guru Bhandari, Ashique Rahman Nessa Moon, Uzma Sabir Bhat, Mashhoor Shahid, Markus Helender, and Jukka K. Nurminen. CVEfixes: Automated collection of vulnerabilities and fixes for open-source software. In *Proceedings of the 18th International Conference on Mining Software Repositories (MSR)*, 2021.

Taolue Chen, Alejandro Flores-Lamas, Matthew Hague, Zhiyang Han, Denghang Hu, Shuang Kan, Anthony W. Lin, Giles Reger, and Naijun Wu. Solving string constraints with regex-dependent functions through transducers with priorities and variables. *Proceedings of the ACM on Programming Languages*, 6(POPL), 2022.

Xiarun Chen, Qien Li, Zhou Yang, Yongzhi Liu, Shaosen Shi, et al. Vulchecker: Achieving more effective taint analysis by identifying sanitizers automatically. In *2021 IEEE 20th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*, 2021.

Wen-Hao Chiang, Peixuan Li, Qiang Zhou, Subarno Banerjee, Martin Schaef, Yingjun Lyu, Hoan Nguyen, and Omer Tripp. Inference for ever-changing policy of taint analysis. In *Proceedings of the 46th International Conference on Software Engineering: Software Engineering in Practice*, ICSE-SEIP '24, pp. 452–462, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400705014. doi: 10.1145/3639477.3639738. URL https://doi.org/10.1145/3639477.3639738.

Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2008.

Jonah Ghebremichael, Saastha Vasan, Saad Ullah, Greg Tystahl, David Adei, Christopher Kruegel, Giovanni Vigna, William Enck, and Alexandros Kapravelos. Multi-agent taint specification extraction for vulnerability detection, 2026. URL https://arxiv.org/abs/2601.10865.

Zinan Gu, Haoxiang Yan, and Peisen Yao. Pyflow: An inter-procedural static analysis framework for python. arXiv:2608.07026, 2026.

Pieter Hooimeijer, Benjamin Livshits, David Molnar, Prateek Saxena, and Margus Veanes. Fast and precise sanitizer analysis with BEK. In *22nd USENIX Security Symposium (USENIX Security 11)*, 2011.

Yuchen Ji, Hongchen Cao, and Jingzhu He. Staint: Detecting second-order vulnerabilities in PHP applications with LLM-assisted bi-directional static taint analysis. In *40th IEEE/ACM International Conference on Automated Software Engineering (ASE 2025)*, 2025.

David Klein, Thomas Barber, Souphiane Bensalim, Ben Stock, and Martin Johns. Hand sanitizers in the wild: A large-scale study of custom javascript sanitizer functions. In *2022 IEEE 7th European Symposium on Security and Privacy (EuroS&P)*, pp. 236–250, 2022.

Xuan-Bach Le et al. Graphledger: Repository-level vulnerability detection via graph-based compression and assumption validation. In *41st IEEE/ACM International Conference on Automated Software Engineering (ASE 2026)*, 2026.

Abderahman Lekssays, Luigi Cerullo, et al. LLMxCPG: Context-aware vulnerability detection through code property graph. In *34th USENIX Security Symposium (USENIX Security 25)*, 2025.

Ziyang Li, Daoyuan Zou, Shouhuai Xu, Xinping Ou, Hai Jin, Shanshan Wang, Zhengzi Deng, and Yuelong Zhong. IRIS: LLM-assisted static analysis for detecting security vulnerabilities. *arXiv preprint arXiv:2406.18573*, 2024.

Puzhuo Liu, Chengnian Sun, Yaowen Zheng, Xuan Feng, Chuan Qin, Yuncheng Wang, Zhenyang Xu, Zhi Li, Peng Di, Yu Jiang, and Limin Sun. Llm-powered static binary taint analysis. *ACM Trans. Softw. Eng. Methodol.*, 34(3), February 2025. ISSN 1049-331X. doi: 10.1145/3711816. URL https://doi.org/10.1145/3711816.

Zhuohang Liu, Zhi Wang, Haotong Liu, and Wanpeng Li. An empirical study on static application security testing (sast) tools for python. In *48th IEEE/ACM International Conference on Software Engineering (ICSE 2026)*, 2026.

V. Benjamin Livshits and Monica S. Lam. Finding security vulnerabilities in java applications with static analysis. In *14th USENIX Security Symposium (USENIX Security 05)*, Baltimore, MD, July 2005. USENIX Association. URL https://www.usenix.org/conference/14th-usenix-security-symposium/finding-security-vulnerabilities-java-applications-static.

Hui Meng, Linlin Mao, and Juzhi Peng. Sanitize processing and recognition method driven by large language model. *Netinfo Security*, (12):1990–1998, 2025.

Meta/Pysa developers. Pysa: A security-focused Python taint analyzer built on Pyre. https://github.com/facebook/pyre-check, 2025.

National Institute of Standards and Technology. CVE-2025-61911 detail. National Vulnerability Database, 2025. URL https://nvd.nist.gov/vuln/detail/CVE-2025-61911.

National Institute of Standards and Technology. CVE-2026-47392 detail. National Vulnerability Database, 2026. URL https://nvd.nist.gov/vuln/detail/CVE-2026-47392.

James Newsome and Dawn Xiaodong Song. Dynamic taint analysis for automatic detection, analysis, and signaturegeneration of exploits on commodity software. In *Proceedings of the 12th Annual Network and Distributed System Security Symposium (NDSS '05)*, 2005.

Elie Rizk, Firas Ben Hmida, and Birhanu Eshete. Taintradar: Semantic-aware taint-style vulnerability detection via augmented code property graphs, 2026. URL https://arxiv.org/abs/2607.16456.

Semmle/QL developers. CodeQL: Semantic queries and analyses for code security. https://github.com/github/codeql, 2025. GitHub; CLI 2.22.2 with bundled Python queries.

Youkun Shi, Yuan Zhang, Tianhao Bai, Feng Xue, Jiarun Dai, and Fengyu Liu. XSSky: Detecting XSS vulnerabilities through local path-persistent fuzzing. In *34th USENIX Security Symposium (USENIX Security 25)*, 2025.

H. Su et al. A sanitizer-centric analysis to detect cross-site scripting in php programs. In *2022 IEEE 33rd International Symposium on Software Reliability Engineering (ISSRE 2022)*, 2022.

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 (ICLR 2026)*, 2026a.

Wenhan Wang, Kaibo Liu, An Ran Chen, Ge Li, Zhi Jin, Gang Huang, and Lei Ma. Python symbolic execution with LLM-powered code generation. *arXiv preprint arXiv:2409.09271*, 2024.

Wenhan Wang, Kaibo Liu, Zeyu Sun, An Ran Chen, Ge Li, Gang Huang, and Lei Ma. Can large language models reason about complex execution paths? an empirical study on python. *ACM Transactions on Software Engineering and Methodology*, 2026b.

Xinyi Wang, Yeting Li, Jie Lu, Shizhe Cui, Shizhe Cui, Qin Mai, Yunpei Zhang, Yang Xiao, Feng Li, and Wei Huo. ZIPPER: Static taint analysis for PHP applications with precision and efficiency. In *34th USENIX Security Symposium (USENIX Security 25)*, pp. 8273–8291, Seattle, WA, August 2025a. USENIX Association. ISBN 978-1-939133-52-6. URL https://www.usenix.org/conference/usenixsecurity25/presentation/wang-xinyi.

Ziliang Wang et al. VulAgent: A hypothesis-validation based multi-agent framework for vulnerability detection. *arXiv preprint arXiv:2509.11523*, 2025b.

Yichen Xie and Alex Aiken. Static detection of security vulnerabilities in scripting languages. In *15th USENIX Security Symposium (USENIX Security 2006)*, 2006.

Fangzhi Xu, Zhiyong Wu, Qiushi Sun, Siyu Ren, Fei Yuan, Shuai Yuan, Qika Lin, Yu Qiao, and Jun Liu. Symbol-LLM: Towards foundational symbol-centric interface for large language models. In Lun-Wei Ku, Andre Martins, and Vivek Srikumar (eds.), *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 13091–13116, Bangkok, Thailand, August 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.acl-long.707. URL https://aclanthology.org/2024.acl-long.707/.

A Overhead and Case Studies

Overhead is gated by design: only units whose paths traverse sanitizer calls pay for the adjudication layer. On the expert core with a fully warm response cache, adjudication adds 0.4 s per unit on average (max 1 s); across the campaign the shared cache held 1,852 entries, so the four-arm comparisons of Figure 4 pay each LLM call once.

Case study: CVE-2025-61911 (python-ldap)(National Institute of Standards and Technology, 2025). The sanitizer escape_filter_chars conditions its escaping on input type and mode. On the vulnerable pole, the type condition is attacker-violable with witness, no guard rescues, and the flow is judged real (TP); on the fixed pole the added unconditional entry check is extracted as a guard satisfying Eq. equation 2 for every violating value, and the same call now blocks (TN). This is clean Z3-plus-guard discrimination with no diff knowledge involved.

Case study: CVE-2018-7753 (bleach). The fixed pole's whitelist rewriter is compiled by the LLM into the closed rewriter vocabulary, differentially executed against the real function on the 255-item corpus (first-round compile misreading octal escapes caught and repaired), and accepted only after structural containment. This step turns a persistent false positive into a TN with an execution transcript as evidence.

Case study: CVE-2026-3644. A predicate-shaped sanitizer core has no internal guards; SanTA synthesizes the caller guard from the taint-path closure. The vulnerable pole's bypass path reaches the dangerous operation outside the closure, so the synthesized guard cannot cover it. The limitation is the vulnerability itself, and the method reports it rather than masking it.

B Condition-Sanitizer Identification across LLM Backends

Purpose. Identification accuracy reflects the method's effectiveness: the more precisely the core sanitizer and its effectiveness conditions are identified, the more the mechanical layer has to verify and the better the resulting detection verdicts. The experiment scores this accuracy directly against ground truth across three LLM backends and asks whether identification accuracy converts into end-to-end detection performance, and through which stage this conversion happens.

Design. Three backends run the frozen protocol with thinking mode enabled: qwen3-next-80b-a3b-instruct, deepseek-v4-flash-0731, and glm-5.3-flash. No existing method exposes this metric, since

taint engines emit no sanitizer identification and LLM-only baselines declare no closed-vocabulary conditions, so the comparison holds the backend fixed and varies only the adjudication division of labor: A1, the engine plus pure-LLM identification and judgment (the non-fusion arm: the LLM identifies the core sanitizer and itself issues the path verdict), and A2, the full method (the fusion arm: the LLM proposes, the mechanical layer decides). The backend-free A0 engine anchors both arms and returns identical scores on every backend. One transport-level adaptation is disclosed: on thinking-mode endpoints reasoning counts against the completion budget, so the transport escalates the budget (to 32k tokens; two deepseek-v4-flash-0731 units required 64k) until a complete JSON answer returns; prompts, temperature, the closed vocabulary, the Z3 timeout, and the engine are unchanged, and the ladder affects only whether an answer physically arrives, not how it is scored. Accuracy is read at three levels from the adjudication records: core identification (flow-level, alias-tolerant match against the GT core function), condition recognition (a condition of the GT kind declared on the GT data parameter, on core-located input-family flows), and verification verdict (decided verdicts against the pole expectation; the status distribution of declared conditions is reported in Figure 3a).

Dataset. The expert core of CVE-SAN-Python (§6.2): 40 standalone pole units carrying the per-CVE ground truth that annotates the core sanitizer function (with aliases), its family, and, for the input family, the condition kind and data parameter the fix semantics imply. The remaining CVEs of the dataset lack condition-level ground truth and enter only the end-to-end evaluations (Table 2).

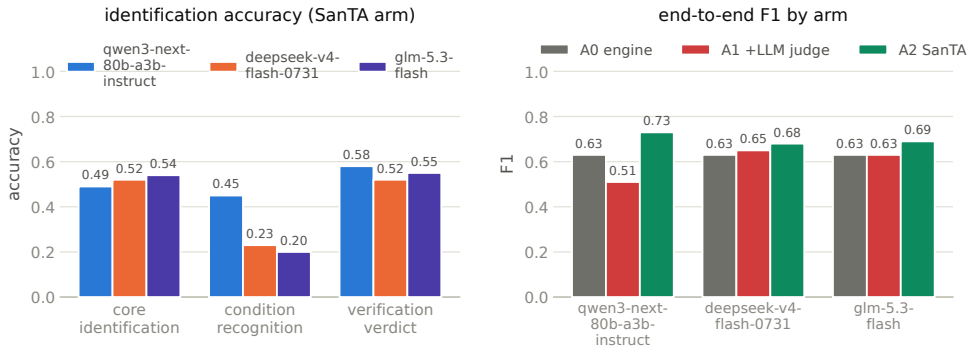


Figure 5: **Condition-sanitizer identification accuracy across LLM backends (CVE-SAN-Python expert core).** Left: three-level accuracy of the SanTA arm per backend, covering core identification (flow level), condition recognition (kind and data parameter), and verification verdict (decided flows). Right: end-to-end F1 of A0 (engine, backend-free), A1 (engine + pure-LLM identification and judgment), and A2 (full SanTA) per backend. The fusion arm A2 is the best configuration on every backend.

Results. Figure 5. Core identification is backend-robust: the GT core is named on 35/71 (0.49), 37/71 (0.52), and 38/71 (0.54) of flows, and the non-fusion arm identifies at the same rates (0.46/0.51/0.55). Naming the sanitizer is a skill every backend carries. The discriminating level is condition recognition: 0.45 (13/29) for qwen3-next-80b-a3b-instruct against 0.23 (7/31) and 0.20 (6/30), and verdict accuracy follows the same ordering ($0.58 > 0.55 > 0.52$). End-to-end, the fusion arm is the best configuration on every backend: A2 F1 0.73/0.68/0.69 against A1's 0.51/0.65/0.63 and the shared A0 anchor at 0.63. Identification without verification never clearly improves on the engine (at best 0.65 against 0.63), and on the strongest condition recognizer it collapses to 0.51, below the engine: unverified sanitizer judgments block paths that carry real vulnerabilities (§6.3). Across backends, A2's F1 ordering equals the verdict-accuracy ordering and is opposite to core identification. The backend with the highest core rate (glm-5.3-flash, 0.54) is not the best detector, and the one with the lowest (qwen3-next-80b-a3b-instruct, 0.49) is. What converts into detection is the accuracy of the recognized condition and of the verdict, not sanitizer naming: identification supplies, verification decides, and it is this neuro-symbolic composition, not the backend's identification skill alone, that turns identification accuracy into detection performance.

C CVEFIXES POOL CONSTRUCTION AND CVE-LEVEL READ-OUT

The held-out pool is built by mechanical gates, all tool-agnostic and applied identically to every arm. From 536 source rows (360 paired fix-rows plus 176 no-diff rows), the funnel drops 101 unusable rows (empty or non-code content), collapses 133 duplicate fix-pairs, and removes 5 no-diff rows overlapping the paired set, leaving 297 snapshots (227 paired, 70 add-only) that aggregate to 163 distinct CVEs.

Read-out. Let $L(v)$ denote the label functions of CVE v—the functions its fix touches, widened to the changed module when the fix is module-level. An arm *fires* on snapshot s if it emits an alert anchored in $L(v)$. Then $\mathrm{TP}(v) = \mathbf{1}[\exists s \in \mathrm{vuln}(v) : \mathrm{fire}(s)]$, $\mathrm{FP}(v) = \mathbf{1}[\exists s \in \mathrm{fixed}(v) : \mathrm{fire}(s)]$, $R = \mathrm{TP}/163$, and $P = \mathrm{TP}/(\mathrm{TP} + \mathrm{FP})$. A CVE can contribute both poles (98 do), which is why $\mathrm{TP} + \mathrm{FP}$ can exceed 163 and why precision and recall use different denominators.

Arm chain. SanTA's CVE-level fire is the conjunction: anchored engine hit $\wedge$ verdict layer not retracted $\wedge$ (fixed pole only) patch-difference module not unflagging $\wedge$ fix-semantics module not unflagging. CodeQL runs in three configurations—stock security suite (F1 0.35), engine-rule-synced (F1 0.49), and rule-synced wide (F1 0.51)—and Table 2 reports the strongest. The stock engine in its clearing configuration was not part of this pool's frozen runs; on the earlier 100-pair CVEfixes subset at unit level it scores 0.48 F1 against SanTA's 0.49.

D DISTRUST ISOLATION AND STATISTICAL UNCERTAINTY

Table 3: **Distrust isolation on the held-out pool (163 CVEs, one read-out).** All arms share anchoring and aggregation; each row adds one gate.

Arm	P	R	F1	TP	FP
Distrust (substrate)	0.50	0.83	0.62	135	137
+ verdict layer	0.50	0.83	0.62	135	137
SanTA (full chain)	0.56	0.83	0.67	135	105
CodeQL (strongest)	0.54	0.48	0.51	78	67

Table 3 separates what the substrate already detects from what each gate adds. Distrust-at-candidates is the engine in SanTA's candidate-marking configuration with every adjudication gate bypassed: any anchored rule match is reported and nothing is ever retracted. Adding the verdict layer alone flips no CVE in either direction on this pool; the precision margin comes from the fixed-pole gates, which unflag 32 false-positive CVEs while leaving the recall column untouched (0 detections retracted). All numbers are recomputed from frozen artifacts with a consistency guard against the main-table read-out.

Significance. Exact McNemar tests on the paired per-CVE outcomes: vulnerable pole, SanTA vs CodeQL-strongest, discordant b/c = 58/1, $p < 0.001$; SanTA vs distrust, b/c = 0/0, p 1.000. On the fixed pole SanTA retains significantly more alerts than CodeQL ($p < 0.001$), which we disclose as the cost of the recall-oriented read-out. Bootstrap 95% confidence intervals over 10,000 CVE resamples (seed fixed): SanTA F1 [0.64, 0.70], recall [0.77, 0.88], precision [0.54, 0.59]; CodeQL-strongest F1 [0.45, 0.56]; distrust F1 [0.60, 0.64]. SanTA's F1 interval is disjoint from both comparators'.

Determinism. Point estimates are single runs: LLM stages run under frozen decoding with response caching (§5), and adjudication is deterministic given a condition set. Bootstrap resampling is seeded and reproducible.

E CVE-SAN-PYTHON: CONSTRUCTION AND ANNOTATION

Construction. Candidate CVEs were collected from NVD/GHSA records of Python projects whose fixes touch a sanitizer-adjacent function. A CVE enters the benchmark if (i) complete vulnerable and fixed versions are available, (ii) a standalone unit per pole reproduces the vulnerable or fixed behavior in isolation, and (iii) the fix semantics hinge on a conditional sanitizer—effectiveness dependent on a runtime-varying input property. Fifty CVEs pass; each contributes two standalone

units (100 total). Every unit carries a meta record with the CVE id, the standalone entry file, and per-pole ground-truth files; the CVE-to-library mapping is listed in Table 4.

Expert core. The 20-CVE expert core is the subset whose fix semantics admit a condition-level annotation: each entry records the core sanitizer function(s) with aliases, the sanitizer family, the fix-implied condition kind, and the data-parameter aliases. Entries were derived from the fix diff and cross-checked against the advisory text; matching at scoring time is alias-tolerant (normalized substring containment). The expert core drives all component metrics (§6.2); the full 50-CVE set drives end-to-end detection only (Table 2).

Disjointness. CVE-SAN-Python and the CVEfixes pool of Appendix C share no CVE.

Table 4: **CVE-SAN-Python: the 50 benchmark CVEs and their libraries.**

CVE	Library	CVE	Library
CVE-2018-7753	bleach	CVE-2020-27783	lxml
CVE-2020-6816	bleach	CVE-2020-6817	bleach
CVE-2021-23980	bleach	CVE-2021-26813	markdown2
CVE-2021-28957	lxml	CVE-2023-23934	werkzeug
CVE-2023-24329	cpython-urllib-parse	CVE-2023-31047	django
CVE-2023-32681	requests	CVE-2023-36053	django
CVE-2023-41164	django	CVE-2023-43665	django
CVE-2024-22195	jinja2	CVE-2024-23334	aiohttp
CVE-2024-24680	django	CVE-2024-34064	jinja2
CVE-2024-34078	python-html-sanitizer	CVE-2024-35195	requests
CVE-2024-37891	urllib3	CVE-2024-4340	sqlparse
CVE-2024-45230	django	CVE-2024-53861	pyjwt
CVE-2024-56326	jinja2	CVE-2024-56374	django
CVE-2025-27516	jinja2	CVE-2025-32873	django
CVE-2025-61911	python-ldap	CVE-2025-61912	python-ldap
CVE-2025-66221	werkzeug	CVE-2025-66418	urllib3
CVE-2026-0672	cpython-http-cookies	CVE-2026-21441	urllib3
CVE-2026-27962	authlib	CVE-2026-28498	authlib
CVE-2026-32597	pyjwt	CVE-2026-34514	aiohttp
CVE-2026-35536	tornado	CVE-2026-3644	cpython-http-cookies
CVE-2026-44307	mako	CVE-2026-44896	mistune
CVE-2026-4519	cpython-webbrowser	CVE-2026-45409	idna
CVE-2026-48526	pyjwt	CVE-2026-48710	starlette
CVE-2026-53878	django	CVE-2026-59894	sqlparse
CVE-2026-59923	mistune	CVE-2026-91991	tornado

F TRUST COMPUTING BASE

Table 5: **Trust computing base beyond the trusted encoder** (Table 1). Each row states the assumption a component carries and the direction in which it fails.

Component	**Trust assumption**	**If the assumption fails**
Fact layer $\mathcal{F}$	pinned types and value sets cover the values reachable at the call site; facts outside the analyzed scope are unavailable	over-covered $\mathcal{F}$ wrongly establishes a condition (FN direction); an uncovered value weakens proofs toward undecided or violated, so the alarm is retained
Guard whitelist	extracted test expressions reject exactly the values they guard	a spurious rescue blocks a real violation (FN direction); a missed guard keeps the alarm (FP direction)
Differential execution	the sandbox and adversarial corpus decide rewriter-model equivalence	an accepted but wrong model blocks a real bypass (FN direction)
Engine rules and anchoring	substrate rules cover the label functions, and anchoring matches the fix locus	both poles affected symmetrically; a read-out property, not an adjudication property

Undecided never blocks: a component that cannot prove its conservative outcome routes the unit to undecided rather than to a block, so most failure directions above degrade toward alarms. The two directions not so guarded—fact-layer over-coverage and accepted rewriter models—are false-negative-direction risks, disclosed rather than masked.

G WILD-CAMPAIGN LEDGER

Table 6 lists the 30 analyzed releases with confirmed findings and reporting channels, and the clean libraries with the reason no finding was submittable. The 15 confirmed findings split into 12 command injections (CWE-78), 2 code injections (CWE-94), and 1 regular-expression denial of service (CWE-1333); every confirmed finding carries its adjudication record—the attacker-violable witness input that satisfies the guarding construct yet still carries the attack, together with the failed guard-correspondence obligation—which is what the vendor reports rest on. The other 15 analyzed releases produced no submittable finding: their execution sinks are absent, admin-gated, vendor-documented, or local-trust. CVE ids are withheld from the double-blind copy and recorded in the frozen submission ledger artifact.

H THE TAINT ENGINE BEHIND SANTA

SanTA's substrate is a Python static taint-analysis engine developed in-house for this work. It provides the capabilities a mainstream taint engine is expected to provide: declarative source/sink/sanitizer rules maintained as snapshots, inter-procedural taint propagation, and enumeration of taint paths from sources to sinks.

Its distinguishing mechanism is the LLM-facing candidate emission mode used throughout this paper (§4.1). In the standard configuration, a sanitizer match clears taint and the path disappears from the output. In candidate-emission mode, a path that matches a sanitizer rule is *not* cleared: the engine outputs the full path unchanged—the complete source-to-sink propagation sequence, the intra-project call chain, and the provenance and downstream use of the sanitizer call's arguments—marked as a sanitizer candidate for downstream analysis. The engine emits every rule-matched path without pre-pruning weak matches, and named rules that never enter this flow keep the engine's original behavior. This guarantees adjudication completeness at the interface: the hypothesis and verification stages operate on the engine's full path surface rather than on fragments left over after rule-based cleansing, and no information is lost between the two layers.

The engine is open source; the complete source code accompanies the submission as supplementary material.

Table 6: **Wild-campaign ledger: 30 analyzed libraries, one pinned release each.** Upper block: libraries with vendor-confirmed vulnerabilities (reporting channel in parentheses). Lower block: libraries with no submittable finding: the sinks are admin-gated, vendor-documented, or local-trust. The pool spans agent and LLM frameworks and widely used packages from web serving, data processing, crawling, and templating. Commit shas are withheld from the double-blind copy.

Package	Release	Finding	CVE id / channel
Confirmed findings (15)			
DeepResearchAgent	main	code inj. (CWE-94)	**CVE-2025-8667** (vuldb)
agentUniverse	v0.0.18	command inj. (CWE-78)	**CVE-2025-8697** (vuldb)
agno	v1.7.5	command inj. (CWE-78)	**CVE-2025-8665** (vuldb)
AdalFlow	v1.0.5a1	command inj. (CWE-78)	(vuldb)
LangBot	v4.1.1	command inj. (CWE-78)	(vuldb)
PraisonAI	v2.2.72	command inj. (CWE-78)	(vuldb)
agentscope	v0.1.6	command inj. (CWE-78)	(huntr)
any-agent	1.0.0	command inj. (CWE-78)	(vuldb)
autogen	python-v0.6.4	code inj. (CWE-94)	(huntr)
langgraph-mcp	main	command inj. (CWE-78)	(vuldb)
letta	0.8.17	command inj. (CWE-78)	(vuldb)
llama-index	v0.12.50	command inj. (CWE-78)	(huntr)
mcp-proxy	v0.8.2	command inj. (CWE-78)	(vuldb)
mcpadapt	v0.1.11	command inj. (CWE-78)	(vuldb)
smolagents	v1.20.0	ReDoS (CWE-1333)	(huntr)
No submittable finding (15)			
litellm	1.84.0		sinks admin-gated / route-unreachable; rewrites hardened, MCP stdio whitelist documented
transformers	4.45.2		interpreter import whitelist kills path
langchain	1.2.12		sinks local-trust / remote-API
openai-agents	0.22.2		sinks L0/by-design
google-adk	2.9.0		local exec self-labeled unsafe
crewai	1.15.21		exec = developer-authored flows
mcp (python-sdk)	2.2.0		stdio list-form exec, by-design
fastmcp	4.0.3		providers import user-owned files; sandbox declared
browser-use	0.13.10		subprocess = tool plumbing, L0
guidance	0.3.1		no untrusted exec sink
requests	2.32.4		no process or code execution sink
flask	3.1.1		no process or code execution sink
pandas	2.2.3		expression evaluation developer-supplied, local-trust
scrapy	2.13.0		no untrusted exec sink
jinja2	3.1.6		sandboxed template evaluation, vendor-documented