Supplementary Material for GraphLedger: Repository-Level Vulnerability Detection via Graph-Based Compression and Assumption Validation

Thi-Hong-Cuc Le
Faculty of Computer Science and Engineering
Ho Chi Minh City University of Technology (HCMUT)
Ho Chi Minh City, Vietnam
Vietnam National University Ho Chi Minh City
Ho Chi Minh City, Vietnam
lthcuc.sdh242@hcmut.edu.vn

Hoang-Quoc-Bao Hua
Faculty of Computer Science and Engineering
Ho Chi Minh City University of Technology (HCMUT)
Ho Chi Minh City, Vietnam
Vietnam National University Ho Chi Minh City
Ho Chi Minh City, Vietnam
hhqbao.sdh241@hcmut.edu.vn

Xuan-Bach Le*
Faculty of Computer Science and Engineering
Ho Chi Minh City University of Technology (HCMUT)
Ho Chi Minh City, Vietnam
Vietnam National University Ho Chi Minh City
Ho Chi Minh City, Vietnam
lexuanbach@hcmut.edu.vn

This document collects the extended appendices for the paper *GraphLedger: Repository-Level Vulnerability Detection via Graph-Based Compression and Assumption Validation* (ASE 2026). References to "Theorem N", "Table N", and section numbers point to the main paper. Appendix letters here match the letters used in the main paper.

A Implemented Vulnerability Types

This appendix lists the vulnerability categories explicitly supported by the current GraphLedger artifact. These are the canonical vulnerability types used by the implementation for source/sink selection, evidence retrieval, dataset mapping, and result reporting.

The implementation also supports common aliases for these categories, including variant names such as `sql_injection`, `path_traversal`, and direct CWE-based inputs such as `CWE-89`, `CWE-22`, `CWE-79`, `CWE-502`, `CWE-862`, `CWE-863`, `CWE-78`, and `CWE-918`. Inputs outside these implemented categories are not handled through specialized source/sink definitions or evidence rules.

How Vulnerability Type Affects Compression. AACC is run separately for each requested vulnerability class. For the selected class, the implementation first resolves aliases to a canonical type, then instantiates class-specific source and sink markers over node text. These markers seed the dependency slices used by PruneNonPreserving. The pruning rule itself is shared across classes: it retains nodes that lie on bounded source–sink dependency slices, nodes in the forward/backward source–sink corridor, explicit source and sink nodes, AST ancestors needed to keep enclosing declarations intact, alias-mediated dependency nodes, and optional behavioral-summary anchors. What changes across vulnerability classes is therefore not the pruning operator, but the source/sink seeds and the security-relevant guard vocabulary that determine which corridor is preserved.

Operational Interpretation. For example, under `sqlinjection`, AACC preserves the corridor from input acquisition to SQL execution sites and tends to keep adjacent nodes that may confirm or refute parameterization or sanitization. Under `authbypass`, by contrast, the same pruning routine is anchored on identity/authority sources and privileged-action sinks, so it keeps nearby authorization checks rather than query-construction logic. The same pattern holds for the remaining classes: the vulnerability type determines which source–sink slices define relevance, and pruning then removes graph regions outside those class-specific dependency closures.

B Baseline Configuration Details

Table 3 summarizes the controlled configurations used for the main baselines. These settings correspond to the shared-protocol evaluation in Section 5.2; the released scripts and configuration files reproduce the same setup.

C Hypothesis Prompting Details

This appendix summarizes the prompting interface used for hypothesis generation. The LLM does not analyze the raw repository directly. Instead, it receives a compressed CPG-derived summary and a target vulnerability type, and returns a bounded structured hypothesis for later verification by the Assumption Ledger.

Prompt Role. The prompt is designed to make the LLM act as a conservative hypothesis generator rather than a free-form vulnerability reporter. Its purpose is to:

- ground generation in the compressed graph summary G',
- require explicit source–sink reasoning,
- force the model to expose the assumptions behind the claim, and
- return machine-readable output for downstream ledger processing.

Inputs. The prompt takes two inputs:

- a CPG summary extracted from the compressed repository graph, and
- a canonical vulnerability type from Appendix A.

The CPG summary contains only repository entities preserved by AACC, so the LLM reasons over dependency-relevant context rather than over the full repository.

Conservative Decision Rule. The prompt explicitly instructs the model to report a vulnerability only when it can identify a concrete

*Corresponding author.

Table 1: Vulnerability types implemented in the current artifact.

Canonical Type	Representative CWE	Description
`sqlinjection`	CWE-89	SQL injection
`pathtraversal`	CWE-22	Path traversal / path injection
`xss`	CWE-79	Cross-site scripting
`deserialization`	CWE-502	Unsafe deserialization
`authbypass`	CWE-862 / CWE-863	Authorization bypass / improper authorization
`commandinjection`	CWE-78	OS command injection
`ssrf`	CWE-918	Server-side request forgery

Table 2: How each implemented vulnerability class specializes AACC compression and pruning.

Type	Compression Seeds (Representative Sources)	Compression Seeds (Representative Sinks)	Guard / Pruning Focus
`sqlinjection`	request parameters, headers, cookies, request bodies, Python request objects, CLI input	JDBC execution calls, statement construction, Python cursor execution, raw SQL dispatch	retain the data-flow corridor into query execution and nearby nodes indicating escaping, parameterization, prepared statements, or sanitization
`pathtraversal`	request paths, URIs, servlet paths, headers, Python path/input objects	file constructors, file streams, path resolution, byte reads/writes, file readers/writers, Python file I/O	retain the path-manipulation corridor and nearby checks for canonicalization, normalization, base-directory validation, or path sanitization
`xss`	request parameters, headers, cookies, attributes, Python request values	response writers, print/write calls, template/render calls, DOM-style output sites	retain the flow into rendered output and nearby encoding/escaping utilities, sanitizers, or HTML-safe wrappers
`deserialization`	input streams, byte buffers, request parameters, object reads	object deserializers, XML decoders, JSON/pickle-style deserialization entry points	retain the flow into object materialization and nearby allowlists, object-input filters, class validation, or deserialization guards
`authbypass`	request parameters, headers, remote-user and principal lookups	privileged actions, admin operations, deletion routines, sensitive execution entry points	retain the access-control corridor and nearby authorization predicates such as role checks, permission checks, and explicit security guards
`commandinjection`	request parameters, headers, cookies, bodies, Python request objects, CLI input	runtime execution APIs, process builders, shell execution calls, Python subprocess and OS command wrappers	retain the flow into command construction/execution and nearby escaping, allowlists, validation, quoting, or replacement-based filtering
`ssrf`	request parameters, headers, query strings, request bodies, Python request objects	URL construction, outbound HTTP clients, URL connection APIs, request libraries, URL open calls	retain the flow into outbound network requests and nearby host allowlists, URL validators, hostname checks, or URL parsing guards

source-to-sink data-flow path in the supplied CPG summary. If the evidence is incomplete or ambiguous, the model must return a negative result rather than speculate. This bias is intentional: the system prefers false negatives at the generation stage because unsupported claims are more costly for repository-scale auditing than missed hypotheses that may be recovered through later analysis.

Vulnerability-Specific Guidance. The prompt includes source, sink, and key-condition guidance for each supported vulnerability class. The canonical type set and representative CWE mappings are listed in Appendix A; the prompt simply instantiates the same source–sink-plus-guard template for the requested class.

Structured Output. The model must return one JSON object containing a Boolean decision, the analyzed vulnerability type, a grounded description, source and sink locations, a stepwise data-flow trace, an ordered list of assumptions, and a bounded confidence score in $[0, 1]$.

Assumption Encoding. The most important prompt constraint is that every assumption must be *falsifiable*. The prompt therefore forbids vague statements such as "this looks dangerous" and instead requires conditions that can later be validated or refuted from repository evidence, such as "no input sanitization occurs between source and sink." Assumptions are listed from fundamental to specific, and an assumption may optionally reference earlier assumptions using dependency indices. This structure allows the Assumption Ledger to reason about assumption dependencies explicitly rather than treating all assumptions as independent.

Table 3: Baseline configuration details.

Method	Configuration	Tuning	Prompt Setup
CodeQL	Security-extended rule pack	dataset-calibrated queries	–
IRIS	Default taint inference	paper settings	task prompt
LLMxCPG	CPG slicing + LLM reasoning	context budget tuning	structured analysis prompt
VulAgent	multi-agent workflow	validation threshold tuning	role-based prompts
Vul-RAG	retrieval + vulnerability KB	top-k retrieval tuning	RAG prompt
gpt-4o-mini	single-agent reasoning	temperature/grid search	analysis prompt
SemTaint	taint-analysis-aware agent	framework-aware heuristics	taint analysis prompt
LLMAO	fault localization via LLM	paper settings	defect localization prompt
RepoAudit	graph traversal + memory-backed reasoning	traversal depth tuning	structured audit prompt
GraphRAG-style retrieval	graph-based RAG retrieval	community detection tuning	vulnerability retrieval prompt

Design Rationale. The prompting strategy is aligned with the formal model in Section 3. The LLM proposes a candidate hypothesis $h = (s, t, \phi, \mathcal{A})$ by identifying a source s, a sink t, a concrete flow description corresponding to ϕ, and a set of semantic assumptions $\mathcal{A}$. The prompt does not ask the model for a final verdict. Instead, it asks for a structured claim whose assumptions can be audited by the ledger. This separation keeps semantic generation and evidence-based verification decoupled.

D Detailed Complexity Analysis

This appendix summarizes the main complexity terms used in the paper. The main point is that GraphLedger combines a potentially expensive graph-construction stage with largely linear downstream compression and ledger passes under bounded search.

CPG Construction. Let n denote repository size, f the number of procedures, and $G = (V, E)$ the resulting repository graph. Building AST and CFG structures is linear in code size, while PDG construction can be quadratic in the size of a procedure under conservative intraprocedural data-flow analysis. With lightweight call-graph recovery, the graph-construction stage is therefore conservatively bounded by

$$O\left(n + \sum_i n_i^2 + f^2\right),$$

which we summarize pessimistically as $O(n^2 + f^2)$ in the worst case. In practice this bound is loose because the constructed graphs are sparse and the frontends use lightweight approximations rather than full whole-program analysis.

AACC Compression. Compression has three relevant costs: dependency slicing, pruning, and optional summary generation. For a bounded set of vulnerability markers, the slicing stage is

$$O(t \cdot (|V| + |E|)),$$

where t is the number of queried sink markers. Pruning itself is linear in the graph size because it is driven by forward/backward reachability and per-node retention checks, and optional behavioral-summary generation is also linear in the retained summary region. Thus AACC is linear in graph size when t is treated as a small constant, and for sparse graphs becomes effectively $O(t \cdot |V|)$.

Ledger Validation. Let h be the number of hypotheses, k the average number of assumptions per hypothesis, a the number of unique assumptions, d the retrieval depth limit, deg the average degree in the dependency view, and c the number of retrieval candidates scored per assumption. The ledger registers hypotheses in $O(h \cdot k)$ time and then performs bounded evidence retrieval and state updates in

$$O(h \cdot k + a \cdot (d \cdot \deg + c)).$$

When k, d, and deg are small constants and c is bounded by the local retrieval neighborhood, ledger validation is close to linear in the number of hypotheses and assumptions.

Bounded Search Parameters. The implementation fixes maximum path length to $\ell = 20$ and path budget to $m = 8$ per source–sink pair in the main evaluation setting. These parameters bound both the path-enumeration component of compression metrics and the bounded exploration used in the formal model. Larger values increase cost monotonically; smaller values improve throughput but may truncate deeper cross-module flows. The reported results should therefore be interpreted under a fixed deployment-oriented search budget rather than exhaustive path exploration.

End-to-End Complexity. Combining graph construction, compression, hypothesis generation, and ledger validation yields the conservative end-to-end bound

$$O\left(n + \sum_i n_i^2 + f^2 + t \cdot (|V| + |E|) + |V'| + h \cdot k + a \cdot (d \cdot \deg + c)\right),$$

where $|V'|$ is the compressed graph size presented to the LLM. In typical repositories, $|V| \propto n$, $f \propto n$, and $|V'|$ is much smaller than $|V|$ after compression, so practical runtime is dominated by graph construction, dependency slicing, and external model latency.

Space Complexity. The dominant memory cost is storing the repository graph itself. For sparse graphs this is $O(|V| + |E|) = O(|V|)$. The compressed graph is a smaller induced subgraph, behavioral summaries add only compact per-procedure metadata, and the assumption ledger requires $O(h \cdot k + a)$ space. Overall memory usage therefore remains linear in repository graph size.

Practical Bottlenecks. In the current implementation the main costs are:

- CPG construction and graph assembly,
- dependency slicing and pruning on large merged repositories,
- external LLM latency during hypothesis generation, and
- evidence retrieval when many assumptions remain unresolved.

Table 4: Complexity comparison with baselines.

Method	Time Complexity	Space Complexity
CodeQL	$O(n^2)$	$O(n)$
LLMxCPG	$O(n \cdot t)$	$O(n)$
VulAgent	$O(n \cdot h \cdot r)^*$	$O(n)$
GraphLedger	$O(n^2 + f^2)$ worst-case†	$O(n)$

$^*r$ is the number of validation rounds and may be unbounded.
†This is a conservative upper bound; practical runtime is often substantially lower under sparse graphs and bounded search.

Comparison with Baselines. GraphLedger therefore has complexity comparable to existing repository-level methods while adding explicit convergence guarantees and conditional path preservation (under the compression assumptions of §3).

E Detailed Proofs

This appendix gives concise proofs of Proposition 1and Theorem 1, making the bounded-search and summary-safety assumptions explicit.

E.1 Detailed Proof of Proposition 1

Proof of Proposition 1. Fix a source–sink pair (s, t) and assume

$$\exists \pi' \in \Pi_{G'}^{(\ell,m)}(s, t).$$

We show that there exists a corresponding dependency path

$$\pi \in \Pi_G(s, t).$$

Each edge of π' is either an ordinary retained edge or a summary edge. For ordinary retained edges, PruneNonPreserving returns an induced subgraph on the retained vertices, so every non-summary edge in G' is already an edge of G. For a summary edge (u, v), assumption (ii) guarantees a concrete dependency path

$$\rho_{u,v} \in \Pi_G(u, v)$$

in the original graph. Replacing every summary edge of π' by its witness path $\rho_{u,v}$ and leaving retained edges unchanged yields, by concatenation, a path from s to t in G.

Pruning cannot create false paths, because assumption (i) only removes vertices outside the retained slices and corridor and introduces no new ordinary edges. Thus the only non-original edges in G' are the path-backed summary edges already handled above. The bound (ℓ, m) restricts which paths in G' are explored, but not the reconstruction argument itself: every explored path in $\Pi_{G'}^{(\ell,m)}(s, t)$ expands to a path in $\Pi_G(s, t)$.

We conclude that

$$\exists \pi' \in \Pi_{G'}^{(\ell,m)}(s, t) \Rightarrow \exists \pi \in \Pi_G(s, t),$$

which is exactly the claimed compression-safety property. □

E.2 Detailed Proof of Theorem 1

Proof of Theorem 1. Let $h = (s, t, \phi, \mathcal{A})$ be a fixed hypothesis, and let $L_0, L_1, \ldots$ be the sequence of ledger states produced as evidence is incorporated over time. By the monotonicity assumption in Section 3, once an assumption becomes *undermined*, it never returns to *pending* or *validated*.

Assume that at some step j there exists an assumption $A_r \in \mathcal{A}$ such that

$$State_{L_j}(A_r) = undermined.$$

Then, by the definition of

$$valid_L(h) \triangleq \forall A_i \in \mathcal{A},\ State(A_i) \neq undermined,$$

we obtain $\neg valid_{L_j}(h)$, hence $Verdict(h, L_j) = Rejected$ by the first branch of the verdict definition.

Consider any later ledger state L_k with $k \geq j$. Because *undermined* is absorbing, the same assumption remains undermined:

$$State_{L_k}(A_r) = undermined.$$

Therefore $\neg valid_{L_k}(h)$ still holds, so again the rejection branch applies:

$$Verdict(h, L_k) = Rejected \qquad \text{for all } k \geq j.$$

Thus once any assumption is undermined, the hypothesis cannot return to *Verified*, regardless of later supporting evidence. Rejection is therefore monotonic under bounded verification. □

F Per-CWE Detection Breakdown

Tables 5 and 6 report per-CWE-class detection results for GraphLedger (full configuration) on each dataset. All metrics treat *Inconclusive* as a negative prediction.

Table 5: Per-CWE detection on CWE-Bench-Java (Full).

CWE Class	P↑	R↑	F1↑	FDR↓	TP
Path Traversal	0.75	0.32	0.44	0.25	38
Cmd. Injection	0.76	0.22	0.34	0.24	26
XSS	0.73	0.18	0.29	0.27	22
Code Injection†	0.83	0.71	0.77	0.17	15
Deserialization	1.00	0.01	0.02	0.00	1
SSRF	1.00	0.01	0.02	0.00	1
SQL Injection	0.00	0.00	0.00	–	0
Auth. Bypass	0.00	0.00	0.00	–	0

† CWE-94; detected via `commandinjection` marker overlap.

Table 6: Per-CWE detection on CVEfixes (Full).

CWE Class	P↑	R↑	F1↑	FDR↓	TP
Auth. Bypass	0.75	0.26	0.38	0.25	12
SQL Injection	0.75	0.19	0.31	0.25	9
XSS	0.70	0.15	0.25	0.30	7
Deserialization	1.00	0.04	0.08	0.00	2
Cmd. Injection	0.50	0.04	0.08	0.50	2
Path Traversal	1.00	0.02	0.04	0.00	1
SSRF	1.00	0.02	0.04	0.00	1

SQL injection and authorization bypass achieve zero recall on CWE-Bench-Java because the benchmark contains *no instances* of these CWE classes: CWE-Bench-Java covers 4 CWE labels—path traversal (CWE-22, 55 repos), cross-site scripting (CWE-79, 31 repos), code injection (CWE-94, 21 repos), and command injection (CWE-78, 13 repos)—while CWE-89 and CWE-862/863 are absent from its repository set.

Cross-CWE Generalization (CWE-94). Code injection (CWE-94) is not one of GraphLedger's seven declared vulnerability types. Nevertheless, the `commandinjection` detector identifies 15 of 21 vulnerable CWE-94 repositories (P = 0.83, R = 0.71) through source/sink marker overlap: code-injection exploits such as expression-language injection and unsafe `Runtime.exec()` calls share the same sink patterns as OS command injection. Only 3 false positives occur on the 21 fixed counterparts (FP rate 14.3%). This cross-CWE detection is not engineered but emerges from the marker-based architecture, demonstrating that AACC's source–sink abstraction generalizes beyond the types for which explicit markers were designed.

Evaluation Protocol Note. Per-CWE metrics use repository-level (type-agnostic) ground truth: each repository is labeled *vulnerable* or *fixed* regardless of which CWE class it represents. Consequently, when the SQL injection detector correctly produces no alert on a repository whose vulnerability is path traversal, the evaluation still counts that repository as a false negative for the SQL injection class. Per-CWE recall should therefore be read as "fraction of *all* vulnerable repositories detected by this specific CWE detector," not as "fraction of SQL injection bugs missed." Aggregate metrics (Table 5) are unaffected because they take the union across all CWE detectors.

Complementary Dataset Coverage. CVEfixes reverses the coverage pattern: with 90 CWE labels it provides the instances that CWE-Bench-Java lacks. Authorization bypass becomes the most-detected class (TP=12), SQL injection achieves P = 0.75, and unsafe deserialization and SSRF each contribute detections. Path traversal drops to a single detection because CVEfixes contains fewer path-traversal instances relative to its total. This complementary coverage motivates the use of both datasets for comprehensive evaluation. Across both datasets, precision remains above 0.70 for all classes with non-trivial TP counts, indicating that the ledger's false-positive control operates consistently regardless of vulnerability type.

G Submitted All-120 CWE-Bench-Java Results

Table 7 reproduces the *submitted* CWE-Bench-Java detection table computed on all 120 repositories (240 paired instances), which *includes* the 20 repositories used for hyperparameter selection. Primary camera-ready metrics use the 100-repository test holdout (Table 5); see the evaluation limitation in §5.1.

H Validation-Only Preset Sensitivity

Table 8 reports the evidence-preset sensitivity restricted to the 20-repository validation holdout (40 paired instances), i.e., the data on which the *balanced* preset was selected. These numbers are recomputed by filtering the archived predictions to the frozen validation slug list; no re-tuning was performed.

Table 7: CWE-Bench-Java detection on all 120 repositories (n=240 paired instances; submitted version).

Method	Precision↑	Recall↑	F1↑	FDR↓
CodeQL [4]	0.68	0.44	0.53	0.32
IRIS [7]	0.56	0.59	0.57	0.44
gpt-4o-mini [8]	0.53	0.66	0.59	0.47
GraphRAG [2]	0.59	0.68	0.63	0.41
Vul-RAG [1]	0.61	0.69	0.65	0.39
LLMAO [10]	0.63	0.66	0.64	0.37
RepoAudit [5]	0.64	0.68	0.66	0.36
SemTaint [3]	0.62	0.70	0.66	0.38
VulAgent [9]	0.64	0.67	0.65	0.36
LLMxCPG [6]	0.67	0.70	0.68	0.33
GraphLedger (Ours)	**0.75**	**0.73**	**0.74**	**0.25**

Table 8: Evidence-preset sensitivity on the 20-repository validation holdout (40 paired instances).

Preset	Precision↑	Recall↑	F1↑	FDR↓
Low (h=5, τ=0.0, any)	0.93	0.65	0.76	0.07
Balanced (h=10, τ=0.3, any)	0.74	0.70	0.72	0.26
High (h=20, τ=0.5, all)	0.58	0.70	0.64	0.42

Table 9: Quantitative breakdown of residual false positives after ledger validation.

Category	Share	Root Cause
Aliasing / Pointer Analysis	33%	Over-approximate data dependencies caused by imprecise alias resolution and indirect object references.
Concurrency / Async Flows	17%	Missing synchronization semantics and asynchronous execution paths not fully captured by static dependency graphs.
Incomplete CPG Extraction	25%	Parsing limitations or missing inter-file dependency edges during graph construction.
Framework Abstractions	25%	Security checks encapsulated within framework utilities or external libraries not explicitly modeled in summaries.

I Split Robustness over 50 Random Holdouts

Because the exact validation slug list used during tuning was not archived (§5.1), we assess sensitivity to the choice of holdout by refiltering the archived predictions over 50 random 20/100 repository splits. GraphLedger F1 ranges 0.72–0.78 (mean 0.74); VulAgent F1 ranges 0.61–0.67 (mean 0.64). GraphLedger achieves higher F1 *and* lower FDR than VulAgent in all 50 splits, and the two F1 ranges do not overlap, indicating that the reported ordering is not an artifact of any particular split. Per-split values are archived in the artifact (`results_full/sensitivity/robustness_50_splits.json`).

J Detailed Failure Analysis

Table 9 summarizes the dominant residual false-positive categories after ledger validation.

K Positioning Against Prior Work

This section provides a detailed technical comparison between GraphLedger and the most closely related systems: LLMxCPG [6] (CPG-based compression), VulAgent [9] (hypothesis validation), and IRIS [7] (hybrid static+LLM analysis). We clarify the specific algorithmic differences, complementary design choices, and empirical advantages of our integrated approach.

K.1 Comparison with LLMxCPG

LLMxCPG [6] introduced CPG-guided slicing for vulnerability detection, achieving 67.8–90.9% context reduction while preserving vulnerability-relevant information. Our AACC mechanism builds on similar principles but differs in three key aspects:

1. Conditional Path Preservation. LLMxCPG employs backward slicing from sinks with empirical validation of compression effectiveness. AACC formalizes compression as a graph-preserving operator $C : G \to G'$ with reachability preservation properties that hold conditional on the compression assumptions of §3(Proposition 1, Section 4.1). This formalization enables:

- Traceability for bounded verification: explored paths in G' correspond to paths in G, given witness-backed summary edges
- An explicit statement of which vulnerability-relevant paths are retained, and under which assumptions
- Compositional reasoning about compression behavior

2. Behavioral Summary Integration. LLMxCPG focuses on structural slicing without semantic abstraction. AACC can optionally integrate behavioral summaries $\Sigma_f = (Pre_f, Effects_f, Post_f)$ that:

- Replace function bodies with conservative effect models
- Preserve dependency information through abstract interpretation
- Enable deeper compression while maintaining semantic fidelity
- Support hybrid static-dynamic validation of summary correctness

3. Verification-Aware Compression. LLMxCPG optimizes for context reduction alone. AACC co-designs compression with the ledger, ensuring that:

- Retained context includes evidence needed for assumption validation
- Behavioral summaries expose security-critical operations (sanitization, authorization)
- Compression preserves not just reachability but also validation-relevant program facts

Empirical Advantages. Table 10 shows that AACC achieves comparable compression ratios to LLMxCPG (71% vs. 65.0–90.9%) while improving detection precision through verification-aware context selection. Integration with the ledger lifts precision from LLMxCPG's 0.67 to 0.75 and reduces FDR from 0.33 to 0.25.

K.2 Comparison with VulAgent

VulAgent [9] introduced multi-agent hypothesis validation where specialist agents formulate vulnerability hypotheses with explicit assumption conditions, followed by a validator agent that tests these hypotheses. GraphLedger differs in three fundamental ways:

Table 10: Technical comparison: AACC vs. LLMxCPG.

Aspect	LLMxCPG	AACC
Compression basis	Backward slicing	Graph-preserving operator
Formal guarantees	Empirical validation	Provable preservation
Semantic abstraction	None	Behavioral summaries
Verification coupling	Independent	Co-designed
Evidence retention	Not explicit	Validation-aware

Table 11: Technical comparison: Ledger vs. VulAgent.

Aspect	VulAgent	Ledger
Validation model	Multi-agent iterative	State-monotonic
State reversibility	Allowed	Forbidden
Evidence basis	Agent negotiation	Rule-based checks
Convergence	Not guaranteed	Guaranteed
Compression	Independent	Co-designed
Formal properties	None	Theorem 1

1. State-Monotonic Reasoning vs. Iterative Validation. VulAgent performs iterative validation where hypotheses can be revised and re-evaluated across multiple rounds. Our ledger enforces *state-monotonic reasoning* (Theorem 1, Section 3):

- Once an assumption is undermined, it cannot be restored
- Dependent hypotheses are permanently invalidated
- Prevents oscillating conclusions and ensures convergence
- Provides formal guarantees on reasoning consistency

Formally, VulAgent allows state transitions:

$$undermined \to pending \to validated$$

across validation rounds. Our ledger enforces:

$$pending \to \{validated, undermined\} \text{ (irreversible)}$$

2. Structured Evidence vs. Agent Negotiation. VulAgent relies on agent-to-agent communication and negotiation to resolve conflicts. The ledger uses primarily rule-based evidence classification:

- Evidence is extracted via bounded dependency traversal
- Classification is rule-based: $\mathcal{E}(A_i, e) \in \{validated, undermined, neutral\}$
- Validation is reproducible and auditable
- No reliance on model-to-model agreement

3. Compression Integration. VulAgent operates over retrieved repository context without compression guarantees. GraphLedger ensures that:

- Validation occurs over dependency-safe compressed graphs
- Evidence retrieval is bounded by preserved structure
- Compression and verification are jointly optimized

Empirical Advantages. Table 11 shows that monotonic reasoning reduces false discovery rate from 0.36 to 0.25 compared to VulAgent while maintaining comparable recall. The rule-based evidence classification improves reproducibility, although optional

Table 12: Technical comparison: GraphLedger vs. IRIS.

Aspect	IRIS	GraphLedger
Primary objective	Specification inference	Hypothesis validation
Focus	API-level rules	Inter-procedural reasoning
Improves	Recall (coverage)	Precision (FDR)
Static analysis role	Augmentation	Validation
Compression	Not addressed	Core component
Verification	Implicit	Explicit ledger

embedding-similarity ranking can still introduce mild variation during retrieval; this remains substantially smaller than VulAgent's agent negotiation variability (std dev 3.2% vs. 0.1%).

K.3 Comparison with IRIS

IRIS [7] combines static analysis with LLM reasoning to infer missing taint specifications for custom APIs, substantially reducing false alarms. GraphLedger addresses a complementary problem space:

1. Specification Inference vs. Hypothesis Verification. IRIS focuses on *augmenting static analysis* by inferring missing specifications:

- Identifies custom sources/sinks not in standard rule sets
- Learns API-level taint propagation semantics
- Enhances static analyzer coverage

GraphLedger focuses on *validating vulnerability hypotheses*:

- Verifies inter-procedural assumptions underlying vulnerability claims
- Enforces consistency across repository-scale reasoning
- Filters unsupported hypotheses through evidence-based validation

2. Complementary Objectives. IRIS improves *recall* by discovering previously unknown vulnerabilities through specification learning. GraphLedger improves *precision* by reducing false positives through assumption validation. These approaches are orthogonal and potentially composable:

- IRIS could provide enhanced source/sink detection
- GraphLedger could validate IRIS-generated hypotheses
- Integration would combine improved coverage with reduced false alarms

3. Scope Differences. IRIS operates at the API specification level, learning taint rules for library interfaces. GraphLedger operates at the inter-procedural reasoning level, validating assumptions about control flow, data flow, and defensive checks across repository-scale dependency graphs.

Empirical Comparison. IRIS achieves moderate recall through specification inference (P=0.56, R=0.59, F1=0.57). GraphLedger achieves precision 0.75, recall 0.73, and F1 0.74 with FDR 0.25 through assumption validation, representing gains of +0.19 precision and +0.17 F1 over IRIS. The two approaches address complementary aspects of the precision-recall trade-off: IRIS focuses on sound specification coverage, while GraphLedger couples structure-preserving compression with explicit ledger verification.

K.4 Synthesis: Integrated Design

Table 13 synthesizes the key distinctions. GraphLedger's contribution is not individual novelty in compression or validation, but rather their *unified integration* with formal guarantees:

- AACC provides provable preservation properties beyond LLMxCPG's empirical slicing
- The ledger enforces monotonic reasoning with convergence guarantees beyond VulAgent's iterative validation
- The co-design ensures verification operates over dependency-safe context
- The integration targets precision improvement, complementing IRIS's recall optimization

Concrete Example: Where GraphLedger Succeeds. Consider a vulnerability hypothesis claiming that user input flows to a database sink without sanitization.

- **LLMxCPG** compresses context but may retain infeasible paths, leading to false positives
- **VulAgent** validates iteratively but may oscillate if agents disagree on sanitization presence
- **IRIS** may miss the vulnerability if the source/sink APIs are non-standard
- **GraphLedger** (1) compresses with provable preservation, (2) validates the "no sanitization" assumption deterministically, (3) permanently invalidates the hypothesis if sanitization is found, ensuring consistent rejection

This example illustrates how integrated design provides advantages beyond component techniques applied independently.

References

[1] Xueying Du, Geng Zheng, Kaixin Wang, Yi Zou, Yujia Wang, Wentai Deng, Jiayi Feng, Mingwei Liu, Bihuan Chen, Xin Peng, Tao Ma, and Yiling Lou. 2026. VulRAG: Enhancing LLM-Based Vulnerability Detection via Knowledge-Level RAG. *ACM Transactions on Software Engineering and Methodology (TOSEM)* (2026), 30 pages. doi:10.1145/3797277 Just Accepted.

[2] Darren Edge, Ha Trinh, Newman Cheng, Joshua Bradley, Alex Chao, Ankit Mody, Sarah Truitt, and Jonathan Larson. 2024. From Local to Global: A Graph RAG Approach to Query-Focused Summarization. arXiv:2404.16130 [cs.CL]

[3] Jonah Ghebremichael, Saastha Vasan, Saad Ullah, Greg Tystahl, David Adei, Christopher Kruegel, Giovanni Vigna, William Enck, and Alexandros Kapravelos. 2026. Multi-Agent Taint Specification Extraction for Vulnerability Detection. arXiv preprint arXiv:2601.10865. arXiv:2601.10865

[4] GitHub. 2026. CodeQL. https://codeql.github.com/. Accessed: 2026-02-25.

[5] Jinyao Guo, Chengpeng Wang, Xiangzhe Xu, Zian Su, and Xiangyu Zhang. 2025. RepoAudit: An Autonomous LLM-Agent for Repository-Level Code Auditing. In *Proceedings of the 42nd International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 267)*. PMLR, Vancouver, Canada, 21083–21100. https://proceedings.mlr.press/v267/guo25n.html

[6] Ahmed Lekssays, Hamza Mouhcine, Khang Tran, Ting Yu, and Issa Khalil. 2025. LLMxCPG: Context-Aware Vulnerability Detection Through Code Property Graph-Guided Large Language Models. In *Proceedings of the 34th USENIX Security Symposium*. USENIX Association, Seattle, WA, USA, 489–507.

[7] Ziyang Li, Saikat Dutta, and Mayur Naik. 2024. LLM-Assisted Static Analysis for Detecting Security Vulnerabilities. arXiv preprint arXiv:2405.17238. arXiv:2405.17238

[8] OpenAI. 2024. GPT-4o System Card. https://openai.com/research/gpt-4o-system-card. Accessed: 2026.

[9] Ziliang Wang, Ge Li, Jia Li, Hao Zhu, and Zhi Jin. 2025. VulAgent: Hypothesis-Validation Based Multi-Agent Vulnerability Detection. arXiv preprint arXiv:2509.11523. arXiv:2509.11523

[10] Aidan Z. H. Yang, Claire Le Goues, Ruben Martins, and Vincent J. Hellendoorn. 2024. Large Language Models for Test-Free Fault Localization. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering (ICSE)* (Lisbon, Portugal). Association for Computing Machinery, New York, NY, USA, Article 17, 12 pages. doi:10.1145/3597503.3623342

Table 13: Positioning synthesis: GraphLedger vs. closely related work.

Capability	LLMxCPG	VulAgent	IRIS	GraphLedger
Structure-preserving compression	✓	×	×	✓
Conditional path preservation (§3)	×	×	×	✓
Behavioral summaries	×	×	×	✓
Hypothesis validation	×	✓	×	✓
Monotonic reasoning	×	×	×	✓
Rule-based evidence classification	×	×	×	✓
Specification inference	×	×	✓	×
Compression-verification co-design	×	×	×	✓
Convergence guarantees	×	×	×	✓
Primary strength	Context reduction	Iterative validation	API coverage	Integrated precision