Reducing False Positives in Vulnerability Detection: A Survey of LLM-Based and Agentic Approaches

Daniil Ogurtsov[1] Igor Rumiantsev[2]
[1]Pashov Audit Group, dan.ogurtsov@pashov.com
[2]Odd Sequence, gorumyantsev@gmail.com

March 2026

Abstract. Security teams face overwhelming false positive rates: traditional SAST tools produce 68–95% false positives, SOC analysts routinely leave the majority of alerts uninvestigated, and analyst burnout is pervasive. This survey analyzes 100 sources (academic papers, vendor data, production case studies) covering 10 distinct AI/LLM-based approaches to false positive reduction in security tooling. We find that no single approach dominates: hybrid SAST+LLM pipelines eliminate 94–98% of false positives in one industrial study on reliability bugs (null pointer, out-of-bounds, divide-by-zero; transferability to security-specific SAST is not yet established); triage memories offer fast ROI (2 memories, 2.8× improvement in one vendor case study); and proof-of-concept validation achieves zero false positives by confirming exploitability. Benchmark-to-production gaps vary by model: in one study ($n = 58$), Gemini-2.5-pro drops from F1 = 0.910 on a synthetic benchmark to F1 = 0.372 on real-world code, while GPT-5 achieves F1 = 0.955 with precision = 1.0 on the same real-world dataset, suggesting the gap is model-specific rather than universal. Production systems that report the best results combine 2–3 methods, though no controlled study has compared combined vs. single-approach effectiveness. We categorize approaches along three axes (when applied, what it needs, guarantee type), compare cost-effectiveness, and identify seven open problems including benchmark decontamination, dataset quality (35–61% of vulnerability labels are incorrect), and business logic vulnerability detection.
Keywords: false positives, static analysis, SAST, large language models, vulnerability detection, alert fatigue, agentic frameworks, code security, triage, benchmark evaluation

1 Introduction

Terminology

Throughout this paper we use the following abbreviations and metrics: **FP** (false positive): a finding reported by a tool that is not a real vulnerability; **TP** (true positive): a correctly identified real vulnerability; **FN** (false negative): a real vulnerability missed by a tool; **Precision**: TP / (TP+FP), the fraction of reported findings that are real; **Recall**: TP / (TP+FN), the fraction of real vulnerabilities that are detected; **F1**: the harmonic mean of precision and recall, $2 \cdot P \cdot R / (P + R)$, a single measure of detection quality; **FDR** (false discovery rate): $1 - \text{Precision}$; **SAST** (Static Application Security Testing): tools that analyze source code without executing it; **LLM** (Large Language Model): neural networks trained on large text corpora, used here for code analysis; **CWE** (Common Weakness Enumeration): a standardized classification of software vulnerability types; **CVE** (Common Vulnerabilities and Exposures): a public identifier for a specific vulnerability; **OWASP**: Open Worldwide Application Security Project, maintains standard benchmarks and vulnerability catalogs; **PoC** (Proof of Concept): a demonstration that a vulnerability is exploitable; **RAG** (Retrieval-Augmented Generation): augmenting LLM inference with external knowledge retrieval; **RL** (Reinforcement Learning): training models via reward signals rather than labeled examples.

Motivation

This survey originates from a practical need. As a Web3 security audit firm, we routinely evaluate AI-assisted tools for smart contract and application security analysis. Understanding which approaches produce reliable results, which overstate their performance, and what the actual cost-effectiveness trade-offs are is essential for integrating these tools into production audit workflows. We found no existing survey that systematically compares available AI/LLM-based false positive reduction methods across both traditional application security and Web3 domains, with attention to the gap between benchmark claims and real-world deployment.

Traditional SAST tools produce 68–95% false positives in production environments. SOC teams receive thousands of alerts daily, investigate fewer than half, and experience high analyst turnover. We analyzed 100 sources (academic papers, vendor data, production case studies) to map currently available AI/LLM-based approaches that address this problem. No single approach dominates. Benchmark-to-production gaps exist but vary significantly by model. Production systems that report the best results tend to combine 2–3 methods. No controlled study has directly compared combined vs. single-approach effectiveness.

Summary of key findings: a Tencent industrial study eliminates 94–98% of FPs in a hybrid pipeline at \$0.0011–\$0.12 per alarm, though on reliability bugs rather than security vulnerabilities [4]. The lowest-cost intervention is Semgrep's triage memories: 2 memories yield a 2.8× improvement in one vendor case study [84]. The benchmark-to-production gap is model-dependent: in the ZeroFalse framework ($n = 58$), Gemini-2.5-pro drops from F1 = 0.910 (OWASP) to F1 = 0.372 (OpenVuln), while GPT-5 achieves F1 = 0.955 with precision = 1.0 on the same OpenVuln dataset [37].

Key Numbers at a Glance

* Tencent study covers reliability bugs (null pointer dereference, out-of-bounds, divide-by-zero), not security vulnerabilities. Transferability to security-specific SAST (injection, XSS, access control) is not established. Source: [4].

Table 1: Key numbers at a glance

Metric	Value
Baseline SAST FP rate	68-95%
Highest FP elimination reported	94-98% (Tencent, reliability bugs) *
Benchmark gap (model-dependent)	F1: 0.910 → 0.372 (Gemini) **
Cheapest per-alarm approach	$0.0011–$0.12/alarm
Fastest ROI (vendor-reported)	2 memories → 2.8x (Semgrep metric)
Wrong vulnerability labels in datasets	35-61% Croft [99]

** Gemini-2.5-pro achieves F1=0.910 on OWASP (synthetic benchmark) but drops to F1=0.372 on OpenVuln (real-vulnerability benchmark with noisy code, n=58). However, GPT-5 achieves F1=0.955 with precision=1.0 (zero false positives) on the same OpenVuln dataset, indicating this gap is model-specific, not universal. Source: ZeroFalse [37].

Research Questions

This survey is structured around five questions:

- **RQ1**: Which AI approaches achieve the highest FP reduction, and under what conditions (industrial vs benchmark)?
- **RQ2**: How large is the gap between benchmark performance and real-world deployment?
- **RQ3**: What are the cost-effectiveness trade-offs across approaches?
- **RQ4**: What evidence quality supports each approach (peer-reviewed, vendor self-reported, single study)?
- **RQ5**: What remains unsolved, and where should research focus next?

RQ1–RQ3 are addressed in the category sections and the Master Comparison Table. RQ4 is addressed in the Evidence Hierarchy. RQ5 is addressed in Open Problems.

2 Overview: The FP Reduction Landscape

The 10 Categories

Approaches to FP reduction cluster into 10 distinct categories, ordered from most empirically validated to most experimental:

1. **Hybrid SAST+LLM Pipelines** - SAST detects, LLM triages/filters
2. **Agentic Frameworks** - autonomous agents with codebase access
3. **Multi-Agent Systems** - specialized collaborating agents
4. **PoC / Runtime Validation** - prove exploitability before reporting
5. **Context Enrichment** - feed LLMs better code context
6. **Path Feasibility / Constraint Solving** - formally prove FP infeasibility
7. **Fine-Tuning & RL** - train models to reduce over-prediction
8. **LLM-Synthesized Analyzers** - LLM builds tools, not analyzes directly
9. **Triage Memories & Feedback Loops** - accumulate knowledge from triage decisions
10. **CWE-Specialized Prompting** - per-vulnerability-type rules and prompts

When Applied: post-scan approaches work with existing SAST output; pre-deployment approaches require upfront investment; runtime approaches improve over time.

Guarantee Type: only Path Feasibility (formal proof) and PoC Validation (confirmed exploit) provide non-probabilistic guarantees. All LLM-based classification is inherently probabilistic.

2.1 Hybrid SAST+LLM Pipelines

What it is

Two-stage pipeline: a traditional SAST tool (Semgrep, CodeQL, Infer, etc.) runs first and generates a list of findings. Then an LLM evaluates each finding in code context and classifies it as TP or FP.

How it works

1. SAST scans codebase and produces warnings with code locations
2. For each warning, extract relevant code context (surrounding code, data flow, function signatures)
3. LLM receives the warning + code context + CWE description
4. LLM reasons about whether the flagged pattern is actually exploitable
5. LLM returns verdict (TP/FP) with explanation

Key results

Strengths

- **Most empirically validated** approach: multiple industrial studies (Tencent [4] with 433 real alarms)
- **Cost-effective**: $0.0011-$0.12 per alarm, vs 10-20 min manual review (Tencent [4])
- **Preserves SAST coverage**: LLM doesn't replace SAST, just triages its output
- **Transparent**: LLM provides reasoning explanation for each classification (Datadog [41])

Limitations

- **Trade-off between TP and FP filtering**: "prompts optimized for catching true positives risk misclassifying false positives" Datadog [41]
- **Backbone-dependent**: weaker LLMs show limited improvement. Claude Sonnet 4 and GPT-5 significantly outperform smaller models (Sifting the Noise [35])
- **CWE-dependent**: injection-style vulns are filtered reliably, but policy- and cryptography-related CWEs are harder (Sifting the Noise [35])
- **Gemini collapse**: Gemini-2.5-pro achieves F1=0.910 on OWASP (synthetic) but collapses to F1=0.372 on OpenVuln (real-vulnerability benchmark with noisy code) (ZeroFalse [37]), showing the gap between synthetic and

Table 2: Baseline false positive rates across tool categories

Context	Baseline FP Rate	Source
Traditional SAST (production code)	68-78% avg, up to 95% in production	Endor Labs [6] (NIST data)
SAST on GitHub repos (overall)	91%+	Ghost Security [1]
Python/Flask command injection SAST	99.5%	SastBench [3]
LLM standalone (project-scale)	85.3% false discovery rate	LLM VD Project Scale [5]
SOC alerts	46-90%	Microsoft/Omdia [17], Torq [22]
AppSec alerts (non-critical)†	95-98%	OX Security [2]
Smart contract tools (Mythril)	~80%	SmartLLM [59] benchmark
Smart contract tools (Slither)	~69%	SmartLLM [59] benchmark
GPT-4 on smart contracts (zero-shot)	~58% (derived: 1 - 42.3% precision)	SmartLLM [59] benchmark

† This figure reflects severity prioritization (only 2–5% of findings are critical), not false positive rate. Low-severity true positives are excluded from the "critical" count but are not false positives.

Table 3: Classification axes for the 10 FP reduction categories

Category	When Applied	What It Needs	Guarantee Type
1. Hybrid SAST+LLM	Post-scan (triage)	SAST output + LLM API	Probabilistic
2. Agentic	Post-scan (triage)	Codebase access + LLM API	Probabilistic
3. Multi-Agent	Post-scan (triage)	Multiple LLM calls	Probabilistic
4. PoC Validation	Post-scan (confirmation)	Sandbox + exploit gen	Deterministic (for confirmed)
5. Context Enrichment	Pre-inference (prep)	Static analysis infra (CPG, CFG)	Probabilistic
6. Path Feasibility	Post-scan (filtering)	SMT solver + SAST paths	Formal proof (when solver succeeds)
7. Fine-Tuning	Pre-deployment (training)	Labeled dataset + GPU	Probabilistic
8. LLM-Synthesized	Pre-deployment (tool creation)	Historical patches	Deterministic (at runtime)
9. Triage Memories	Runtime (accumulating)	Past triage decisions	Rule-based
10. CWE-Specialized	Pre-inference (prompt design)	Security expertise per CWE	Probabilistic

real-world benchmarks is real and model-specific

Experience & insight

Tencent's industrial study [4] is notable for testing on actual enterprise static analysis output (328 FP, 105 TP from 433 real alarms). **Important caveat**: the study covers reliability bugs (null pointer dereference, out-of-bounds access, divide-by-zero), not security vulnerabilities (injection, XSS, access control). Whether the 94–98% FP elimination rate transfers to security-specific SAST categories is not established. Key finding: standalone LLMs with advanced prompting (chain-of-thought, few-shot) still underperform the hybrid combination of LLM + static analysis artifacts. The LLM needs the SA's intermediate results (call graphs, data-flow traces) to reason effectively.

An alternative to snippet-level analysis is providing the LLM with access to the entire codebase.

2.2 Agentic Frameworks

What it is

Instead of feeding an LLM a single code snippet, give an autonomous agent access to the entire codebase. The agent can navigate files, read dependencies, trace data flows, and explore context on its own before making a verdict.

How it works

1. Agent receives a SAST warning (file, line, CWE)
2. Agent has tools: file read, grep, code navigation, AST parsing
3. Agent autonomously explores the codebase to gather evidence
4. Agent traces data flows, checks sanitizers, examines call chains
5. Agent returns TP/FP verdict with evidence trail

Key results

Strengths

- **Highest FP reduction on benchmarks**: >92% down to 6.3% is the best reported number (Sifting the Noise [35])
- **Codebase-aware**: agents can trace actual data flows across files, not just local snippets
- **Self-directed exploration**: can discover context that a static prompt would miss
- **Versatile**: works with any SAST backend

Table 4: Hybrid SAST+LLM pipeline results

System	Eval Type	Baseline FP	After Hybrid	FP Reduction	Recall	Source
Tencent (LLM4SA)	Industrial (433 alarms)	>75%	~2–6%	**94–98%**	maintained	[4]
SAST-Genius	Real-world scan	64.3% (prec. 35.7%)	10.5% (prec. 89.5%)	**91%** (225→20)	maintained	[61]
ZeroFalse	Benchmark (OWASP, OpenVuln)	varies	F1=0.912 / 0.955	significant	>90%	[37]
IRIS (CodeQL+GPT-4)	Real-world (curated)	CodeQL: 27 vulns	IRIS: 55 (+104%)	FDR improved 5pp	doubled	[43]
VulSolver	Benchmark (OWASP)	varies	acc. 97.85%, recall 100%	near-total	100%	[65]

Table 5: Agentic framework results

System	Baseline FP	After Agent	FP Reduction	Notes	Source
Sifting the Noise (best config)	>92% (OWASP)	6.3%	**92%→6.3%**	GPT-5, Claude Sonnet 4 best	Sifting the Noise [35]
RepoAudit	N/A (new detections)	78.43% precision	built-in validator	$2.54/project	RepoAudit [36]
OpenAI Aardvark	N/A	92% detection rate	sandbox validation	10 CVEs assigned	Aardvark [39]
GitHub Taskflow Agent	N/A	~30 real vulns found	MCP multi-agent	open-source	GitHub Taskflow [74]

Limitations

- **Expensive**: requires many LLM calls per finding (agent needs multiple tool calls)
- **Slow**: full codebase exploration takes time
- **Benefits are backbone-dependent**: "agentic frameworks significantly outperform vanilla prompting for stronger models (Claude Sonnet 4, GPT-5)" but benefits are "strongly backbone- and CWE-dependent" (Sifting the Noise [35])
- **Expert oversight still needed**: Intruder [38] found that "expert input is a must" for delivering quality custom checks. Hallucinations remain a problem, expert engineers remain essential

Experience & insight

The "Sifting the Noise" study (Sifting the Noise [35]) is the most systematic comparison. Three frameworks tested (Aider, OpenHands, SWE-agent) with multiple LLM backbones. The results are clear: agentic helps strong models get stronger, but doesn't fix weak models. Also, the improvement is CWE-dependent: injection-style CWEs benefit most, but policy and crypto CWEs remain hard for agents too.

GitHub Security Lab GitHub Taskflow [74] deployed their Taskflow Agent in August 2025 and found ~30 real vulns. They use an MCP-enabled multi-agent framework with YAML-based taskflow grammar that templates the triage process. Key insight: the system targets repetitive FP patterns that are obvious to humans but hard to formalize as traditional rules. The process needs to be structured and repeatable.

A natural extension of single-agent approaches is deploying multiple agents with different specializations.

2.3 Multi-Agent Systems

What it is

Instead of one model or one agent, deploy multiple specialized agents that collaborate. Each agent focuses on a different aspect (detection, validation, reasoning, repair) and they exchange information.

How it works

Varies by system, but common patterns:

- **Detector + Validator** (two-stage): one agent finds issues, another validates AgenticSCR [66]
- **Ensemble voting** (parallel): multiple agents independently assess, majority rules MultiVer [69]
- **Router + Specialists** (hierarchical): router categorizes, specialists analyze per CWE type MulVul [68]
- **Hypothesis-Validation** (iterative): form hypothesis, gather evidence, validate VulAgent [80]

Key results

Strengths

- **Complementary views**: different agents catch different things. MultiVer [69] shows that security + correctness + performance agents together find more than any alone
- **Zero-shot competitive with fine-tuned**: MultiVer is the first zero-shot system to surpass fine-tuned recall on vulnerability detection (82.7% vs 81.3%), though at the cost of lower precision (48.8% vs 63.9%) MultiVer [69]
- **End-to-end automation**: CodeCureAgent [82] doesn't just classify TP/FP but also generates patches (86.3% correct fix rate)
- **Scalable specialization**: MulVul [68] automates prompt engineering per vulnerability category

Table 6: Multi-agent system results

System	Architecture	Key Result	Src
MultiVer	4-agent ensemble (zero-shot)	**82.7% recall** (exceeds fine-tuned GPT-3.5 at 81.3%). Precision: 48.8%	[69]
AgenticSCR	Detector + Validator	**153% more correct review comments** vs static LLM baseline	[66]
MAVUL	Multi-agent contextual	**62%+ higher pairwise accuracy** vs multi-agent; **600%+ vs single-agent**	[67]
MulVul	Router + RAG specialists	Automated per-category prompt generation via cross-model evolution	[68]
VulAgent	Hypothesis-validation loop	**FP rate reduced ~36%** vs SOTA. Fixed-pair detection +246% avg (+450% max)	[80]
CodeCureAgent	Classify + Repair	**96.8% plausible fix rate** across 1,000 SonarQube warnings	[82]
iAudit	Detector + Reasoner + Ranker + Critic	**F1=91.21%** on 263 real smart contract vulns	[49]
LLM-BSCVM	6-agent collaborative	**5.1% FP rate** (vs 7.2% SOTA)	[50]
LLMBugScanner	5-model ensemble voting	~10% hallucination rate, 60% top-5 accuracy	[48]

Limitations

- **Precision-recall tension persists**: MultiVer achieves 82.7% recall but only 48.8% precision vs 63.9% for fine-tuned baselines MultiVer [69]
- **Voting can't fix shared blindspots**: LLMBugScanner [48] found "voting could not correct shared weaknesses" when all models lacked training examples for minority classes (e.g. access control)
- **Complexity**: more agents = more failure modes, harder debugging
- **Hallucination propagation**: if the detector agent hallucinates, the validator may not catch it. ~10% hallucination rate persists even in ensembles LLMBugScanner [48]
- **Justification quality**: iAudit [49] found that even when models correctly identify a vulnerability EXISTS, they often give the WRONG reason. GPT-4 achieves only 30% precision when both decision AND justification must be correct

Experience & insight

The iAudit finding (iAudit [49]) is critical: precision on the decision alone (is this vulnerable?) is much higher than precision on decision+justification (is this vulnerable AND is this the right reason?). This matters because security engineers need correct reasoning, not just correct labels. A model that says "this is vulnerable because of reentrancy" when the real issue is access control is operationally useless even if it correctly flags the function.

Multi-agent approaches show clear gains: MAVUL reports +62% pairwise accuracy over existing multi-agent systems (and 600%+ over single-agent baselines) MAVUL [67]. But the architecture matters. Detector+Validator chains AgenticSCR [66] are simpler and more predictable than full ensemble voting MultiVer [69], which can introduce precision trade-offs.

The approaches above classify findings probabilistically. An alternative is to confirm exploitability directly.

2.4 PoC / Runtime Validation

What it is

Instead of classifying findings theoretically (is this code pattern vulnerable?), prove it: generate a Proof-of-Concept exploit and try to execute it. If the exploit succeeds, the finding is confirmed. If it fails, it's eliminated as FP.

How it works

1. SAST or LLM identifies potential vulnerability
2. System generates exploit code targeting the specific vulnerability
3. Exploit is executed in sandboxed/isolated environment
4. If exploit triggers the vulnerability, finding is confirmed TP
5. If exploit fails, finding is classified as FP

Key results

Strengths

- **Eliminates FPs by definition**: if you can exploit it, it's real. No false positives possible for confirmed exploits
- **Highest confidence findings**: security teams can prioritize confirmed-exploitable issues
- **Catches semantic vulns**: ZAST.AI covers IDOR, privilege escalation, business logic, not just syntax patterns ZAST.AI [72]
- **Real-world validated**: 119 CVEs is not a toy benchmark ZAST.AI [72]

Limitations

- **Not all vulns are exploitable in isolation**: some vulnerabilities require specific deployment conditions, multi-step attack chains, or user interaction that can't be replicated in a sandbox
- **Blind to latent vulnerabilities**: if a vulnerability isn't currently exploitable but could become exploitable after a config change, PoC validation would miss it
- **False negatives risk**: inability to generate exploit $\neq$ not vulnerable. Exploit generation is itself an undecidable problem
- **Expensive and slow**: generating and executing PoCs is computationally heavy
- **Web3-specific advantage**: blockchain execution is deterministic and self-contained, making PoC validation easier than for web apps. Hypernative [95] achieves $<$0.001% FP for runtime monitoring, but that's transaction monitoring, not code auditing

Experience & insight

ZAST.AI [72] is the most prominent example of this approach. The company raised $6M based on a "zero false

Table 7: PoC and runtime validation results

System	Approach	Key Result	Source
ZAST.AI	Auto-generate PoC + execute	**119 CVEs assigned in 2025 with zero false positives**. Azure SDK, Apache Struts, Alibaba Nacos, Langfuse, Koa	ZAST.AI [72]
A1 (Gervais, Zhou)	Agentic exploit on blockchain	**63% success on VERITE benchmark** (36 contracts). Up to $8.59M per exploit. Simply prompting LLMs generates "unverified speculations with high FP rates"	A1 [92]
Alert or Noise	Active behavioral probes	**~93% of "publicly exposed" alerts actually secured** by permissions, auth, bucket policies	Alert or Noise [29]
Anthropic SCONE-bench	Agent exploits smart contracts	**51.11% exploit rate** (207/405 contracts, $550M simulated). Post-cutoff: 19/34 (55.8%), $4.6M. 2 zero-days in 2,849 fresh contracts	SCONE-bench [47]
OpenAI Aardvark	Sandboxed validation	**92% detection rate**, validates by attempting to trigger vulns in sandbox. 10 CVE assignments	Aardvark [39]

positive" value proposition and reported 119 CVEs in production projects including Microsoft, Apache, and Alibaba. The key innovation: automated PoC generation + automated execution + only reporting what's exploitable.

A1 [92] provides the counterpoint for smart contracts: "simply prompting LLMs generates unverified vulnerability speculations with high false-positive rates." A1 validates through actual blockchain execution (fork the chain, run the exploit). 63% success rate means 37% of potentially real vulns can't be automatically exploited, which could be FNs.

The Alert or Noise study (Alert or Noise [29]) is underappreciated. They applied active behavioral probes to cloud security alerts and found ~93% were false alarms, the "exposed" resources were actually protected by permissions and auth controls. This is PoC validation applied to infrastructure alerts rather than code.

PoC validation provides high confidence but is computationally expensive. A lower-cost alternative is to improve the context provided to the LLM.

2.5 Context Enrichment

What it is

Improve LLM vulnerability detection by feeding the model better, more relevant code context. Instead of analyzing one function in isolation, provide the model with dependency information, call chains, data flow graphs, and related code.

How it works

Different approaches to constructing context:

- **Code Property Graphs (CPG)**: combine AST, CFG, PDG into a unified graph, extract relevant slices LLMxCPG [62]
- **Inter-procedural context profiling**: extract callers, callees, globals via static analysis CPRVul [70]
- **Repository-level context**: analyze the whole repo structure, not just the target function VULPO [64]
- **Knowledge-level RAG**: retrieve vulnerability patterns from historical data at inference time Vul-RAG [75]

Key results

Strengths

- **Addresses root cause**: the CORRECT framework [30] demonstrates that "most false positives stem from reasoning errors" that are themselves caused by insufficient context. Fix the context, fix the reasoning
- **Small models become competitive**: VULPO [64] makes a 4B parameter model achieve parity with DeepSeek-R1 (150x larger) through better context and training
- **CPG slicing is efficient**: LLMxCPG [62] reduces code by 67-91% while preserving vulnerability context. Less code = fewer tokens = cheaper + faster
- **Resilient to adversarial manipulation**: LLMxCPG [62] is "resilient to syntactic modifications" because it focuses on semantic structure, not surface patterns

Limitations

- **Context construction is itself expensive**: building CPGs, resolving inter-procedural dependencies, and extracting relevant slices requires significant static analysis infrastructure
- **Context $\neq$ understanding**: CPRVul [70] found that "gains emerge only when processed context is paired with structured reasoning." Raw context without reasoning guidance doesn't help
- **Context window limits**: project-scale analysis requires "hundreds of thousands to millions of tokens" (LLM VD Project Scale [5]), exceeding current context windows

Experience & insight

This may be the most underrated finding in the entire analysis. The CORRECT framework [30] provides the most rigorous analysis of context's role. They tested 13 LLMs and found that community beliefs about LLM unreliability "are artifacts of context-deprived evaluations." When you give LLMs enough context, they achieve 0.8 precision. When you don't, they look terrible. This is a fundamental insight: most negative LLM security results may be measuring context quality, not model capability.

RAG consistently outperforms fine-tuning in this domain RAG Empirical [83]: accuracy 0.86 vs lower for SFT and dual-agent. This suggests that the bottleneck is knowledge access, not model capability. External knowledge retrieval

Table 8: Context enrichment results

System	Context Method	Key Result	Source
LLMxCPG	CPG-based slicing	Code reduced by **67-91%** while preserving vuln context. **F1 improved 15-40%** over baselines	LLMxCPG [62]
CPRVul	Inter-procedural caller/callee/globals	**67.78% accuracy** on PrimeVul (up from 55.17%, +22.9%). Gains ONLY when context + structured reasoning	CPRVul [70]
VULPO	Repo-level RL optimization	**Qwen3-4B-VULPO F1 improves 85%** over base, comparable to DeepSeek-R1 (150x larger)	VULPO [64]
Vul-RAG	Knowledge-level RAG	**16-24% accuracy increase**. 10 unknown Linux kernel bugs (6 CVEs)	Vul-RAG [75]
RAG empirical	RAG vs SFT vs dual-agent	**RAG highest overall: accuracy 0.86, F1 0.85**	RAG Empirical [83]
CORRECT framework	Sufficient context evaluation	With sufficient context, LLMs achieve **precision ∼0.8, F1 ∼0.7** on key CWEs	CORRECT [30]

(vulnerability databases, CWE descriptions) provides more lift than training the model itself.

Context enrichment improves reasoning quality. For certain classes of FPs, formal methods can provide mathematical proof of infeasibility.

2.6 Path Feasibility & Constraint Solving

What it is

Use formal methods or LLM-guided reasoning to determine whether the execution path leading to a flagged vulnerability is actually feasible. If the path constraints are unsatisfiable (the path can never execute), the warning is provably a false positive.

How it works

1. SAST produces a warning with an associated execution path
2. Extract path constraints (conditions that must be true for the path to execute)
3. Feed constraints to an SMT solver (Z3) or LLM-guided reasoner
4. If constraints are unsatisfiable, the path is infeasible and the warning is a proven FP
5. LLMs help by interpreting complex code semantics that formal solvers struggle with

Key results

Strengths

- **Formal guarantees**: when an SMT solver proves a path is infeasible, the FP elimination is mathematically certain, not probabilistic
- **Very high FP filtering**: LLM4PFA achieves 72-96% FP filtering (LLM4PFA [63]), competitive with hybrid approaches
- **Low false negatives**: LLM4PFA only missed 3 out of 45 true positives (LLM4PFA [63]). VulSolver achieved 100% recall on OWASP VulSolver [65]
- **Complementary to SAST**: works specifically on the artifacts SAST already produces (paths, constraints)

Limitations

- **Scalability**: constraint solving is NP-hard in the general case. Complex inter-procedural paths can overwhelm solvers
- **Heuristic fallback**: when Z3 fails, LLM interprets the error and adjusts constraints iteratively LLM4PFA [63], meaning it's heuristic when formal methods fail
- **Path sensitivity only**: doesn't address FPs caused by imprecise pointer analysis, missing specs, or incorrect vulnerability patterns
- **Requires quality SAST output**: if SAST doesn't provide path information, this approach can't work

Experience & insight

LLM4PFA [63] is notable because it combines the best of both worlds: formal methods (Z3 solver) for provable FP elimination, and LLMs for the cases where formal methods fail. When Z3 reports an error, the LLM interprets the error message and adjusts the constraints iteratively. This LLM-guided formal verification is a promising pattern.

VulSolver [65] formulates vulnerability detection as constraint solving with "transfer constraints" (how tainted data flows) and "trigger constraints" (what conditions trigger the vuln). This decomposition is elegant and achieves near-perfect results on OWASP, though real-world performance is likely lower.

Formal proofs are elegant but don't scale to every case. Training the model itself is a more direct alternative.

2.7 Fine-Tuning & Reinforcement Learning

What it is

Instead of using general-purpose LLMs, train models specifically for vulnerability detection. Fine-tune on labeled vulnerability datasets, or use reinforcement learning to optimize for balanced detection (reducing over-prediction while maintaining recall).

How it works

- **Supervised Fine-Tuning (SFT)**: train on labeled TP/FP examples
- **RL with rewards**: define reward signals for correct detection, correct localization, and correct reasoning. Optimize policy through PPO, DPO, or GRPO [78]
- **Cold-start + RL**: initial SFT from teacher model traces, then RL refinement VULPO [64]
- **Multi-stage fine-tuning**: separate Detector and Reasoner stages iAudit [49]

Table 9: Path feasibility and constraint solving results

System	Method	Key Result	Source
LLM4PFA	LLM agent + Z3 solver	Filters **72-96% of FPs**, outperforms baselines by **41-106%**. Only 3 real bugs missed out of 45 TPs	LLM4PFA [63]
VulSolver	Transfer+trigger constraints	**97.85% accuracy, 100% recall** on OWASP. 15 unknown high-severity vulns (CVSS 7.5-9.8)	VulSolver [65]
AdaTaint	Counterfactual path validation	**43.7% FP reduction** vs CodeQL, Joern, LLM-only. Recall +11.2%	AdaTaint [81]

Key results

Strengths

- **Dramatic improvement over base models**: LLM-BSCVM [50] improves accuracy to 91.11%, surpassing untuned CodeLlama 13B by ˜48pp
- **Small models become viable**: VULPO [64] makes a 4B model competitive with DeepSeek-R1 (150x larger). Massive cost reduction
- **Balanced detection**: GRPO [78] specifically targets the over-prediction problem, reducing FPs without proportional recall loss
- **Domain specialization**: general LLMs lack security domain knowledge. Fine-tuning on vulnerability data fills that gap

Limitations

- **Dataset quality bottleneck**: Croft [99] showed only 39-65% of "vulnerable" labels in major datasets are correct. Fine-tuning on noisy labels propagates noise
- **Catastrophic forgetting**: fine-tuning for one CWE may degrade performance on others
- **Repair quality lags**: LLM-BSCVM [50] found that only ˜21% of repaired contracts passed independent verification. Detection improves faster than repair
- **Minority class problem**: LLMBugScanner [48] found minority classes (e.g. access control) remain hard even with fine-tuning. You can't learn patterns from insufficient examples
- **Generalization gap**: models fine-tuned on one dataset may not transfer to real-world codebases with different patterns

Experience & insight

VULPO [64] demonstrates the most impressive result: a 4B parameter model achieving parity with DeepSeek-R1 (150x larger) through RL + repository-level context. The key innovation is multi-dimensional rewards: not just "did you get the answer right?" but also "did you correctly localize the vulnerability?" and "is your reasoning relevant?" This prevents reward hacking where models learn shortcuts.

This creates a circular problem: we are training detectors on data that is 35-61% wrong. The dataset quality problem (Croft [99]) is the fundamental upstream bottleneck. If 35–61% of training labels are incorrect, no amount of fine-tuning can produce reliable detectors. This is why FORGE [53] is important: it automates dataset construction, potentially fixing the training data problem at scale.

Rather than training the model to analyze code directly, an alternative is to have it synthesize specialized analyzers.

2.8 LLM-Synthesized Analyzers

What it is

Instead of using LLMs to analyze code directly (which produces hallucinations), use LLMs to write specialized static analysis checkers. The LLM's role is to synthesize the tool, not to run the analysis.

How it works

1. Collect historical bug patterns from patches and commits
2. LLM studies the bug pattern and generates a specialized static analyzer (checker)
3. Multi-stage validation: run the checker against the original patch to verify it detects the bug
4. Iterative refinement: if the checker produces FPs, LLM adjusts it
5. Deploy validated checker as a traditional static analysis rule

Key results

Strengths

- **Best of both worlds**: LLM creativity for pattern recognition + deterministic static analysis for execution. No hallucinations at runtime
- **Provably effective checkers**: each checker is validated against the original bug before deployment
- **Scalable**: once synthesized, checkers run as fast as traditional static analysis
- **Real-world impact**: 30 CVEs in the Linux kernel is a top-tier result. These are bugs that existed for an average of 4.3 years KNighter [71]
- **Reduces FPs through validation loop**: iterative refinement specifically targets FP elimination in the checker design

Limitations

- **Only works for pattern-matchable bugs**: business logic vulnerabilities, design flaws, and novel attack vectors can't be captured by static checkers
- **Single study**: KNighter is the only published system using this approach. Needs more validation
- **Requires high-quality historical patches**: the approach depends on having examples of fixed bugs to learn from
- **Checker maintenance**: code evolves, and synthesized checkers may become outdated

Table 10: Fine-tuning and reinforcement learning results

System	Method	Key Result	Source
VULPO	RL + repo context	**4B model achieves parity with DeepSeek-R1** (150x larger). 85% F1 improvement over base	VULPO [64]
GRPO	Group Relative Policy Optimization	Addresses over-prediction of certain CWEs, improves balanced detection	GRPO [78]
iAudit	Two-stage fine-tuning	**F1=91.21%** on smart contracts (GPT-4: 30% precision when decision+justification both required)	iAudit [49]
LLM-BSCVM	Fine-tuned CodeLlama + agents	**FP rate: 5.1%** (vs 7.2% SOTA). Accuracy 91.11% (+˜48pp over untuned)	LLM-BSCVM [50]
SmartLLM	Fine-tuned LLaMA + RAG	Accuracy 70%, Recall 100%, Precision 62.5% (vs Mythril 80% FP rate, Slither 69% FP rate)	SmartLLM [59]
LLMBugScanner	Fine-tuned 5-model ensemble	Top-5 accuracy: 60%. 19% improvement over single-model baselines	LLMBugScanner [48]

Table 11: LLM-synthesized analyzer results

System	Method	Key Result	Source
KNighter	LLM synthesizes Linux kernel checkers	**92 new critical bugs** (avg 4.3yr latent). 77 confirmed, 57 fixed, **30 CVEs**	KNighter [71]

Experience & insight

This is the only approach in our survey where the LLM never touches production code. KNighter [71] presented at SOSP'25 (top systems conference) and represents a genuinely novel paradigm. The insight is that LLMs are better at understanding patterns and writing rules than at reliably analyzing code at scale. By separating "understanding the vulnerability pattern" (LLM strength) from "reliably scanning millions of lines" (static analysis strength), you avoid the hallucination problem entirely.

The 4.3-year average latency of discovered bugs is striking: these are deeply hidden bugs that human reviewers and existing tools missed for years. The LLM's ability to synthesize novel detection patterns from historical examples appears to be genuinely additive to human capability.

Synthesizing new analyzers is effective but requires significant upfront effort. A simpler alternative is to accumulate and reuse past triage decisions.

2.9 Triage Memories & Feedback Loops

What it is

Systems that learn from past triage decisions. When a security engineer marks a finding as FP, the system remembers this decision and applies it to similar future findings. Over time, the system builds institutional knowledge about what constitutes a real vulnerability in this specific codebase.

How it works

- **Triage memories**: store human decisions as rules that the LLM retrieves for similar future findings Semgrep Memories [84]
- **Alert tuning rules**: ML models learn which alert patterns are consistently false alarms and auto-suppress them MS Defender XDR [76]
- **Feedback integration**: user agreement/disagreement feeds back into the model's confidence calibration (Semgrep Assistant [42])

Key results

Strengths

- **Improves over time**: each triage decision makes future triage better. Compound returns
- **Respects organizational context**: what's a FP in one codebase may be a TP in another. Memories capture this nuance Semgrep Memories [84]
- **Minimal startup cost**: Semgrep shows that just 2 memories can achieve 2.8x improvement Semgrep Memories [84]
- **High user trust**: 96% agreement rate from security researchers (Semgrep Assistant [42]) means the system aligns with expert judgment
- **Production-proven**: Semgrep (1M weekly scans, multiple enterprises), Microsoft Defender XDR (shipped Jan 2026)

Limitations

- **Cold start**: new codebases have no memories yet. Initial FP rates remain high
- **Bias propagation**: if early triage decisions are wrong, those errors propagate
- **Memory management**: as memories accumulate, they may conflict or become outdated
- **Vendor lock-in**: memories are specific to the platform (Semgrep memories don't transfer to Datadog)

Experience & insight

If we had to recommend one starting point for most teams, this would be it. Semgrep's Memories (Semgrep Memories [84]) is the most elegantly simple approach in this entire analysis. The insight: most FPs in a given codebase are repetitive. The same code pattern triggers the same false alarm in dozens of places. One memory can eliminate hundreds of FPs. The Fortune 500 case study is compelling: 2 memories added, 2.8x improvement in FP detection, immediate impact.

Table 12: Triage memories and feedback loop results

System	Method	Key Result	Source
Semgrep Memories	Store triage decisions as retrievable rules	Without memories: 18 FPs identified. With memories: **580 FPs filtered** on single rule (2.8× improvement per Semgrep's metric, with just 2 memories [84]; note: another Semgrep post [42] reports "5 memories" for the same case study). The 2.8× is Semgrep's internal metric; the raw ratio of filtered findings is 580/18 = 32×.	Semgrep Memories [84]
Semgrep Assistant	AI triage with user agreement	**~20% of findings filtered at 95% agreement**. 96% agreement from security researchers	Semgrep Assistant [42]
Microsoft Defender XDR	ML incident scoring	Scores 0-100 using MITRE ATT&CK, asset criticality, rarity. **18 built-in tuning rules**. Agentic SOC model	MS Defender XDR [76]
Legit Security	ML on 10K+ labeled samples	**86% FP reduction** in secrets scanning with negligible TP impact. Continual learning	Legit Security [77]

Microsoft Defender XDR [76] takes a different approach: instead of remembering individual decisions, it trains ML models on aggregate signal patterns (MITRE ATT&CK correlation, asset criticality, rarity). This is more automated but less transparent. The 18 built-in tuning rules represent Microsoft's accumulated institutional knowledge about what SOC alerts are reliably false.

While triage memories accumulate knowledge over time, CWE-specialized prompting encodes expert knowledge from the outset.

2.10 CWE-Specialized Prompting

What it is

Instead of using one generic prompt for all vulnerability types, create specialized prompts for each CWE category. Each prompt encodes domain-specific knowledge about that vulnerability type, including common patterns, typical false positive scenarios, and verification rules.

How it works

1. For each CWE (e.g., CWE-79 XSS, CWE-89 SQLi), create a specialized prompt template
2. Prompt includes: vulnerability definition, declarative rules, common FP patterns for that CWE
3. When SAST flags a finding, route it to the CWE-specific prompt
4. LLM applies the specialized rules to classify TP/FP

Key results

Strengths

- **Highest reported precision-recall balance**: ZeroFalse F1=0.955 is among the best results in the field (ZeroFalse [37])
- **Encoding expert knowledge**: declarative rules capture security researcher expertise in a reusable format
- **Predictable behavior**: each CWE has known rules, making the system more debuggable than end-to-end approaches
- **Complementary**: can be combined with any other approach (hybrid, agentic, multi-agent)

Limitations

- **Manual effort for rule creation**: writing declarative rules per CWE requires security expertise
- **MulVul's automation**: MulVul [68] addresses this by automating prompt generation through cross-model evolution (Claude proposes, GPT-4o evaluates), but this is still research-stage
- **CWE coverage gap**: only works for vulnerability types with well-defined patterns. Business logic vulns don't map cleanly to CWEs
- **Benchmark vs real-world**: ZeroFalse shows F1=0.955 on OpenVuln benchmark but "Gemini-2.5-pro collapses on real-world noise (F1: 0.372)" (ZeroFalse [37]). The gap between curated benchmarks and production code is enormous

Experience & insight

ZeroFalse [37] provides the clearest evidence that CWE specialization matters. When you give the LLM per-CWE rules instead of generic security prompts, precision and recall both improve significantly. The declarative rules act as "guardrails" that prevent the common LLM failure modes for that specific vulnerability type.

This result highlights the risk of evaluating tools solely on benchmark numbers. Gemini-2.5-pro drops from F1=0.910 on OWASP (synthetic) to F1=0.372 on OpenVuln (real-vulnerability benchmark). This is the most important cautionary finding across all approaches. It demonstrates that synthetic benchmark results can be dramatically inflated, and that robustness to noisy, real-world code is a separate challenge from detecting vulnerabilities in clean test cases.

3 Cross-Cutting Analysis

This table highlights why "zero false positives" is not achievable as a general goal. Each approach trades coverage for precision: PoC validation has zero FPs but misses unexploitable-today-but-vulnerable-tomorrow cases. CWE-specialized prompting works for well-defined patterns but not for business logic. No single approach, and no known combination, covers all six gaps.

What Actually Works Best: The Evidence Hierarchy

Ranked by strength of evidence and real-world validation:

Tier 1: Industrial validation with production data.

1. Hybrid SAST+LLM: 94–98% FP elimination at Ten-

Table 13: CWE-specialized prompting results

System	Method	Key Result	Source
ZeroFalse	CWE-specific prompts + flow-sensitive traces	**F1=0.912 (OWASP), F1=0.955 (OpenVuln)** with per-CWE declarative rules	ZeroFalse [37]
MulVul	Cross-model prompt evolution	Router predicts CWE category, specialized detectors per type. **Automated prompt generation** via Claude→GPT-4o fitness evaluation	MulVul [68]

Table 14: Recommended approach by scenario. These pairings are based on approach characteristics and available evidence, not on controlled comparisons between approaches for each scenario.

Scenario	Best Approach(es)	Why	Evidence
CI/CD SAST triage (high volume, fast turnaround)	Hybrid SAST+LLM, Triage Memories	Fast, cheap ($0.0011/alarm), maintains SAST coverage	Tencent [4], Semgrep Memories [84]
Security audit (high stakes, thoroughness > speed)	PoC Validation, Agentic Frameworks	Confirmed exploitability = highest confidence	ZAST.AI [72], RepoAudit [36]
SOC alert triage (thousands/day, burnout risk)	Triage Memories, ML Scoring	Learns from history, auto-suppresses noise	MS Defender XDR [76], Semgrep Memories [84]
Smart contract auditing (DeFi, high value)	Fine-Tuned + Multi-Agent, PoC (blockchain execution)	Domain-specific training critical. Deterministic execution enables PoC	iAudit [49], SCONE-bench [47]
Secrets scanning (binary TP/FP, validatable)	PoC Validation (active secret validation)	Can literally test if the secret works	Legit Security [77]
Legacy codebase (millions LOC)	LLM-Synthesized Analyzers, Hybrid	LLMs can't process entire codebases; synthesized checkers scale	KNighter [71]
New project, few examples	CWE-Specialized Prompting, RAG	Expert rules don't need training data. RAG provides external knowledge	ZeroFalse [37], RAG Empirical [83]

cent [4] (reliability bugs; see caveat in Section 2.1).
2. Triage Memories: 95%+ categorization accuracy, 96% security researcher agreement [84, 42] (single vendor, self-reported).
3. PoC Validation: 119 CVEs with zero FPs [72] (vendor-reported, zero FPs is by definition since only confirmed exploits are reported).

Tier 2: Strong benchmark results with some real-world deployment.

4. Agentic Frameworks: 92%→6.3% FP on OWASP [35].
5. CWE-Specialized Prompting: F1=0.955 on benchmarks [37].
6. Fine-Tuning: F1=91.21% on smart contracts [49].

Tier 3: Promising research results, further validation needed.

7. Path Feasibility: 72–96% FP filtering [63].
8. Multi-Agent: +62% pairwise accuracy over existing multi-agent [67].
9. LLM-Synthesized Analyzers: 92 kernel bugs, 30 CVEs (SOSP'25) [71].
10. Context Enrichment: +22.9% accuracy [70].

Note: Tier 1 entries have the strongest production evidence but rely on vendor-internal or self-reported data without independent replication. Tier 2-3 entries are independently verifiable academic results but tested primarily on benchmarks.

Key Takeaways

1. **No single approach dominates**. Hybrid SAST+LLM has the most industrial data (though on reliability bugs, not security vulnerabilities); PoC validation has the strongest theoretical guarantee; triage memories have the lowest marginal cost.
2. **Combination is the observed pattern in production**. Most deployed systems (Semgrep, Endor Labs, Aikido) combine at least 2-3 approaches. This is an observational finding from studying production architectures, not a result from controlled experiments comparing combined vs single approaches. Single-approach systems exist but tend to be narrower in scope.
3. **Benchmark-to-production gaps exist but are model-specific**. In the ZeroFalse study (n=58) [37], Gemini-2.5-pro drops from F1=0.910 on OWASP (synthetic) to F1=0.372 on OpenVuln (real-world noisy code), while GPT-5 achieves F1=0.955 with precision=1.0 on the same OpenVuln dataset. This suggests the gap is not universal but depends on model capability. SastBench shows 99.5% FP in Python/Flask SAST [3]. These findings caution against evaluating approaches solely on synthetic benchmarks. This caveat applies to several categories in this survey: Agentic, Multi-Agent, Path Feasibility, and Context Enrichment results are primarily benchmark-derived.

Table 15: Cost comparison across approaches

Approach	Cost per Finding	Speed	Compute Requirements	Source
Hybrid SAST+LLM	$0.0011-$0.12	seconds per alarm	1 LLM call per finding	Tencent [4]
Agentic Framework	higher (multiple tool calls)	minutes per finding	Multiple LLM calls + tool execution	RepoAudit [36] ~$2.54/project
PoC Validation	high (exploit generation + execution)	minutes-hours	Sandbox environment required	ZAST.AI [72]
Fine-Tuning	training cost upfront, low inference	fast inference	GPU for training, then standard inference	VULPO [64]
Triage Memories	low (rule lookup, no LLM call per finding)	instant (rule lookup)	Retrieval system only	Semgrep Memories [84]
CWE-Specialized Prompting	same as base LLM call	same as base LLM call	Standard LLM inference	ZeroFalse [37]

Table 16: Combination strategies observed in production systems

Combination	Example	Why It Works
SAST + LLM Triage + Memories	Semgrep Assistant [42], Semgrep Memories [84]	SAST provides coverage, LLM triages, memories accumulate knowledge
SAST + Path Feasibility + LLM	Tencent [4], LLM4PFA [63]	SAST detects, formal methods prove FPs, LLM handles cases formal methods can't
Multi-Agent + Fine-Tuning	iAudit [49]	Fine-tuned Detector + LLM-agent Ranker/Critic for quality control
SAST + PoC Validation	ZAST.AI [72]	SAST identifies candidates, PoC confirms exploitability
CPG Context + LLM + Ensemble	LLMxCPG [62]	CPG reduces code to relevant slices, LLM analyzes, multiple views validate
Hybrid + CWE-Specialized	ZeroFalse [37]	SAST detects, CWE-specific prompts guide LLM reasoning

4. **For prompting/agentic approaches, backbone strength is critical**. Agentic frameworks help GPT-5 and Claude Sonnet 4 dramatically, but benefits are "backbone- and CWE-dependent" (Sifting the Noise [35]). However, fine-tuning can compensate for weaker backbones (see #7).
5. **Triage memories offer high ROI for low effort** Semgrep Memories [84]. In a Fortune 500 case study, 2 memories produced 2.8x improvement in FP filtering. Low marginal cost makes this an easy starting point, though evidence is limited to Semgrep's self-reported data.
6. **PoC validation provides the highest confidence but is hardest to generalize**. It works perfectly for deterministic environments (blockchain, secrets validation) but struggles with deployment-dependent web vulnerabilities.
7. **Fine-tuning can compensate for model size**. A fine-tuned 4B model matching DeepSeek-R1 (150x larger) VULPO [64] suggests smaller fine-tuned models may be more cost-effective than larger general models for security tasks, provided training data quality is high (see #8).
8. **Dataset quality is the upstream bottleneck**. 35-61% of vulnerability labels are wrong Croft [99]. Fixing training data (FORGE [53]) may matter more than improving detection algorithms.

Open Problems

Seven directions where the field needs work, based on gaps identified across all 10 categories:

1. **Controlled combination studies**. No source in this survey directly compares "combined approach A+B" vs "best single approach A" with the same dataset and methodology. The recommendation to combine 2-3 approaches is based on observation of production architectures, not experimental evidence.
2. **Production-scale evaluations beyond vendor self-reporting**. All three Tier 1 results (Tencent, Semgrep, ZAST.AI) rely on vendor-internal data. Independent reproduction of these results on comparable industrial codebases would significantly strengthen the evidence base.
3. **Benchmark decontamination and real-world noise**. In ZeroFalse [37] (n=58), Gemini-2.5-pro drops from F1=0.910 on OWASP (synthetic) to F1=0.372 on OpenVuln (real-world noisy code), though GPT-5 performs well on both. This suggests synthetic benchmarks like OWASP may be insufficient for evaluating production readiness, particularly for weaker models. Benchmarks that include dead code, irrelevant findings, and noisy context are needed.
4. **Cross-CWE robustness**. Most approaches work well for injection-style CWEs but struggle with policy, cryptography, and access control vulnerabilities (Sifting the Noise [35]). Per-CWE evaluation should be standard, not optional.
5. **Dataset quality at scale**. 35-61% of vulnerability labels are wrong (Croft [99]). FORGE [53] automates dataset construction, but whether automated labeling solves or propagates the quality problem is unproven.

Table 17: Unsolved problems across all 10 approaches

Gap	Why It's Hard	Which Approaches Come Closest
Business logic vulnerabilities	No CWE pattern, no signature, requires understanding application intent	PoC validation (if exploitable), but can't detect design-level flaws
Novel/zero-day attack classes	LLMs can only recognize patterns seen in training data or retrieved context	LLM-Synthesized Analyzers (can learn from patches), but only post-discovery
Multi-step attack chains	Require reasoning across multiple services, configs, and time	Agentic frameworks (closest), but no system reliably chains multi-service reasoning
Configuration-dependent vulnerabilities	Same code is vulnerable or safe depending on deployment environment	PoC validation (tests actual config), but requires access to production-like environment
Supply chain / transitive dependencies	Vulnerability may be in a nested dependency never called by the app	None. Dependabot [14] generates massive FPs here because it checks existence, not reachability
Adversarial evasion	Attackers can craft code to evade AI detectors	LLMxCPG [62] is resilient to syntactic manipulation, but semantic evasion remains open

Table 18: Master comparison of all 10 FP reduction categories. Metrics are not directly comparable across rows; each uses the metric most relevant to that approach.

#	Category	Best Result	Eval Type	Best Source	Recall Maintained?	Production Ready?	Cost/Finding
1	Hybrid SAST+LLM	94-98% FP elimination (reliability bugs)	Industrial	Tencent [4]	maintained	Yes (Tencent, Datadog)	$0.0011-$0.12
2	Agentic Frameworks	92%→6.3% FP rate	Benchmark (OWASP)	Sifting the Noise [35]	Backbone-dependent	Partial (GitHub)	~$2.54/project
3	Multi-Agent	FP reduced 36%, +62% pairwise acc. (avg)	Benchmark	VulAgent [80], MAVUL [67]	Varies	No	Higher
4	PoC/Runtime	Zero FPs (confirmed only)	Real-world (119 CVEs)	ZAST.AI [72]	N/A (FN risk)	Yes (ZAST.AI)	High
5	Context Enrichment	F1 +15-40%, +22.9% accuracy	Benchmark	LLMxCPG [62], CPRVul [70]	Improved	No	Medium
6	Path Feasibility	72-96% FP filtered	Benchmark	LLM4PFA [63]	93.3% (3/45 missed)	No	Medium
7	Fine-Tuning/RL	FP 5.1%, F1 91.21%	Benchmark	LLM-BSCVM [50], iAudit [49]	High	Partial	Training upfront
8	LLM-Synthesized	92 bugs, 30 CVEs	Real-world (Linux kernel)	KNighter [71]	N/A (new detections)	Partial (patches accepted)	Low (at inference)
9	Triage Memories	2.8x improvement, 95% accuracy	Industrial (single vendor)	Semgrep Memories [84]	Yes	Yes (Semgrep, MS)	Low marginal
10	CWE-Specialized	F1=0.955 (OpenVuln, GPT-5), 0.372 (OpenVuln, Gemini)	Benchmark	ZeroFalse [37]	>90% (benchmark)	Partial	Same as base LLM

6. **Business logic and design-level vulnerabilities**. All 10 categories focus on pattern-matchable code vulnerabilities. No current approach reliably detects business logic flaws, design-level weaknesses, or vulnerabilities arising from the interaction of multiple correct components.
7. **Longitudinal decay**. How do these approaches perform over time as codebases evolve, new vulnerability classes emerge, and LLM capabilities shift? No study in this survey measures performance degradation over periods longer than a single evaluation snapshot.

4 Methodology

Scope. This survey covers AI/LLM-augmented approaches to false positive reduction in security tooling (post-2024). Classical formal methods with decades of production history in FP reduction (abstract interpretation, symbolic execution, model checking, traditional taint analysis refinement) are outside scope. Tools such as Astrée, KLEE, Polyspace, and Facebook Infer have strong FP-reduction track records in safety-critical domains but are not covered here, as the survey focuses on approaches that use LLMs or other recent AI techniques.

Search strategy. Sources were collected during March 2026 through systematic search of the following databases and repositories: *arXiv* (39 papers, primary source for preprints in cs.CR and cs.SE), *ACM Digital Library* (8 papers, including ICSE, FSE, SOSP, ACM TOSEM proceedings), *NDSS Symposium proceedings* (2 papers), *ScienceDirect / Elsevier* (1 paper), and *Preprints.org* (1 paper). Industry sources were collected from vendor engineering blogs with empirical data (Semgrep, Datadog, Microsoft, Endor Labs, OpenAI, Anthropic, GitHub, and others) and independent industry reports (SANS Institute, Black

Duck, OX Security, Arctic Wolf). Search queries included combinations of: "false positive reduction," "SAST LLM," "vulnerability detection AI," "alert fatigue SOC," "agentic security," and "smart contract AI audit."

Inclusion criteria. Sources were included if they: (1) present empirical data on false positive rates, precision/recall, or FP reduction metrics; (2) describe a concrete tool, system, or approach (not purely theoretical); and (3) were published after June 2024. Ten older sources (pre-2025) are included when they are explicitly cited by three or more recent sources as foundational work.

Source composition. The final corpus comprises 100 sources: 48 peer-reviewed academic papers, 25 industry reports and surveys, 15 technical benchmarks, and 12 vendor blogs with quantitative data. Of these, approximately 67 contain direct empirical FP data (measured rates, before/after comparisons, precision/recall); the remaining 33 provide industry context, alert fatigue surveys, and theoretical foundations. Ninety sources are post-June 2025; 10 are cited classics.

Data extraction. Where vendor-reported numbers could not be independently verified, this is noted explicitly. Where sources conflict (e.g., Semgrep reports "2 memories" in one post [84] and "5 memories" in another [42] for the same case study), both figures are cited.

Limitations. No results were independently reproduced as part of this analysis; we report what sources claim. Approaches that were attempted but not published, tools that were discontinued, and negative results that never reached publication are not represented. This creates an inherent survivorship bias toward approaches that showed positive outcomes.

The bibliography below includes only the sources directly cited in the article text. The full list of all 100 sources with URLs, authors, dates, and key findings is available in the supplementary materials.

Data collection period: March 2026.

References

[1] Ghost Security, "Exorcising the SAST Demons," Jun 2025. https://www.helpnetsecurity.com/2025/06/19/traditional-sast-tools/

[2] OX Security, "2025 Application Security Benchmark Report," 2025. https://www.ox.security/ox-2025-application-security-benchmark-report/

[3] Feiglin, Dar, "SastBench: A Benchmark for Testing Agentic SAST Triage," Jan 2026. https://arxiv.org/abs/2601.02941

[4] Du, Feng, Zou et al. (Tencent), "Reducing False Positives in Static Bug Detection with LLMs: An Empirical Study in Industry," Jan 2026. https://arxiv.org/abs/2601.18844

[5] Multiple authors, "LLM-based Vulnerability Detection at Project Scale: An Empirical Study," Jan 2026. https://arxiv.org/abs/2601.19239

[6] Endor Labs, "Introducing AI SAST That Thinks Like a Security Engineer," Nov 2025. https://www.endorlabs.com/learn/introducing-ai-sast-that-thinks-like-a-security-engineer

[14] Filippo Valsorda, "Turn Dependabot Off / Dependabot as Noise Machine," Feb 2026. https://www.theregister.com/2026/02/24/github_dependabot_noise_machine/

[17] Microsoft / Omdia, "Unify Now or Pay Later: New Research Exposes the Operational Cost of a Fragmented SOC," Feb 2026. https://www.microsoft.com/en-us/security/blog/2026/02/17/unify-now-or-pay-later-new-research-exposes-the-operational-cost-of-a-fragmented-soc/

[22] Torq, "Cybersecurity Alert Management 2026," Jan 2026. https://torq.io/blog/cybersecurity-alert-management-2026/

[29] Multiple authors, "Alert or Noise? Active Behavioral Analysis for Cloud Security," Aug 2025. https://arxiv.org/html/2508.12584v1

[30] Li et al., "Everything You Wanted to Know About LLM-based Vulnerability Detection But Were Afraid to Ask," 2025. https://arxiv.org/abs/2504.13474

[35] Xiong, Zhang, "Sifting the Noise: LLM Agents in Vulnerability False Positive Filtering," Jan 2026. https://arxiv.org/abs/2601.22952

[36] Guo et al., "RepoAudit: Autonomous LLM-Agent for Repository-Level Code Auditing," ICSE Jul 2025. https://arxiv.org/abs/2501.18160

[37] Iranmanesh et al., "ZeroFalse: Improving Precision in Static Analysis with LLMs," Oct 2025. https://arxiv.org/abs/2510.02534

[38] Intruder, "The State of Agentic AI in Vulnerability Management," Sep 2025. https://www.intruder.io/research/the-state-of-agentic-ai-in-vulnerability-management

[39] OpenAI, "Introducing Aardvark: Agentic Security Researcher," Oct 2025. https://openai.com/index/introducing-aardvark/

[41] Datadog, "Using LLMs to Filter Out False Positives from Static Code Analysis," Oct 2025. https://www.datadoghq.com/blog/using-llms-to-filter-out-false-positives/

[42] Semgrep, "Announcing AI Noise Filtering and Triage Memories," 2025. https://semgrep.dev/blog/2025/announcing-ai-noise-filtering-and-triage-memories/

[43] Li, Dutta, Naik, "IRIS: LLM-Assisted Static Analysis for Detecting Security Vulnerabilities," ICLR 2025 / Jul 2025 v2. https://arxiv.org/abs/2405.17238

[47] Anthropic, "SCONE-bench: AI Agents Find $4.6M in Smart Contract Exploits," Dec 2025. https://red.anthropic.com/2025/smart-contracts/

[48] Zhang et al. (Georgia Tech), "LLMBugScanner: LLM-based Smart Contract Auditing," Dec 2025. https://arxiv.org/abs/2512.02069

[49] Ma et al., "iAudit: Fine-Tuning and LLM-based Agents for Smart Contract Auditing," ICSE 2025. https://dl.acm.org/doi/10.1109/ICSE55347.2025.00027

[50] Multiple authors, "LLM-BSCVM: Vulnerability Management Framework," 2025. https://arxiv.org/abs/2505.17416

[53] Multiple authors, "FORGE: LLM-driven Framework for Large-Scale Dataset Construction," ICSE 2026. https://arxiv.org/abs/2506.18795

[59] Bhat et al., "SmartLLM: Custom Generative AI for Smart Contract Auditing," 2025. https://arxiv.org/abs/2502.13167

[61] Agrawal, Ahi, "SAST-Genius: A Hybrid Static Analysis Framework," Sep 2025. https://arxiv.org/abs/2509.15433

[62] Lekssays et al., "LLMxCPG: Context-Aware Vulnerability Detection Through CPG-Guided LLMs," USENIX Security 2025. https://arxiv.org/abs/2507.16585

[63] Multiple authors, "LLM4PFA: Minimizing False Positives via LLM-Enhanced Path Feasibility Analysis," 2025. https://arxiv.org/abs/2506.10322

[64] Multiple authors, "VULPO: Context-Aware Reasoning for Vulnerability Detection via On-Policy LLM Optimization," Nov 2025. https://arxiv.org/abs/2511.11896

[65] Li, Su et al., "VulSolver: Vulnerability Detection via LLM-Driven Constraint Solving," Sep 2025. https://arxiv.org/abs/2509.00882

[66] Multiple authors, "AgenticSCR: Autonomous Agentic Secure Code Review," Jan 2026. https://arxiv.org/abs/2601.19138

[67] Li, Joshi, Wang, Wong, "MAVUL: Multi-Agent Vulnerability Detection via Contextual Reasoning," Oct 2025. https://arxiv.org/abs/2510.00317

[68] Multiple authors, "MulVul: Retrieval-Augmented Multi-Agent Code VD via Cross-Model Prompt Evolution," Jan 2026. https://arxiv.org/abs/2601.18847

[69] Rajan, "MultiVer: Zero-Shot Multi-Agent Vulnerability Detection," Feb 2026. https://arxiv.org/abs/2602.17875

[70] Multiple authors, "CPRVul: Context-Aware Reasoning for Inter-Procedural Vulnerability Detection," Feb 2026. https://arxiv.org/abs/2602.06751

[71] Multiple authors, "KNighter: Transforming Static Analysis with LLM-Synthesized Checkers," SOSP Aug 2025. https://arxiv.org/abs/2503.09002

[72] ZAST.AI, "ZAST.AI: Zero False Positive AI-Powered Code Security," Feb 2026. https://thehackernews.com/2026/02/zastai-raises-6m-pre-to-scale-zero.html

[74] GitHub Security Lab, "AI-Supported Vulnerability Triage with the GitHub Security Lab Taskflow Agent," Aug 2025. https://github.blog/security/ai-supported-vulnerability-triage-with-the-github-security-lab-taskflow-agent/

[75] Multiple authors, "Vul-RAG: Enhancing LLM-based VD via Knowledge-level RAG," ACM TOSEM 2025. https://dl.acm.org/doi/10.1145/3797277

[76] Microsoft, "Microsoft Defender XDR: ML-Powered Alert Tuning," Jan 2026. https://redmondmag.com/articles/2026/01/09/new-microsoft-machine-learning-tools.aspx

[77] Legit Security, "Using AI to Reduce False Positives in Secrets Scanners," 2025. https://www.legitsecurity.com/blog/using-ai-to-reduce-false-positives-in-secrets-scanners

[78] Multiple authors, "Improving LLM Reasoning for VD via Group Relative Policy Optimization," Jul 2025. https://arxiv.org/abs/2507.03051

[80] Wang et al., "VulAgent: Hypothesis-Validation Multi-Agent Vulnerability Detection," Sep 2025. https://arxiv.org/abs/2509.11523

[81] AdaTaint team, "AdaTaint: LLM-Driven Adaptive Source-Sink Identification and FP Mitigation," Nov 2025. https://arxiv.org/abs/2511.04023

[82] Joos, Bouzenia, Pradel, "CodeCureAgent: Automatic Classification and Repair of SA Warnings," Sep 2025. https://arxiv.org/abs/2509.11787

[83] Multiple authors, "Empirical Evaluation of RAG, SFT, and Dual-Agent Systems for VD," Jan 2026. https://arxiv.org/abs/2601.00254

[84] Semgrep, "Making Zero False Positive SAST a Reality with AI-Powered Memory," 2025. https://semgrep.dev/blog/2025/making-zero-false-positive-sast-a-reality-with-ai-powered-memory/

[92] Gervais, Zhou, "AI Agent Smart Contract Exploit Generation," Jul 2025. https://arxiv.org/abs/2507.05558

[95] Hypernative, "The State of Web3 Security for 2026," Jan 2026. https://www.hypernative.io/blog/the-state-of-web3-security-for-2026-winning-the-red-queen-race-in-cryptos-breakout-year

[99] Croft, Babar (U. Adelaide), "Data Quality for Software Vulnerability Datasets," 2023. https://dl.acm.org/doi/abs/10.1109/ICSE48619.2023.00022