Innovation and Integrative Research Center Journal
ISSN: 2584-1491 | www.iircj.org
Volume-4 | Issue-5 | May-2026 | Page 86-96

AI-Powered CI/CD Security Compliance Checker: A Multi-Layer Detection Architecture Combining Rule-Based Analysis and Large Language Model Semantic Reasoning

[1]Aayushman Singh Thakur, [2]Dr. Jyoti Singh, [3]Dr. Poonam Mishra
[1]Student, [2]Assistant Professor, [3]Associate Professor
[1,2,3]Department of Computer Science & Engineering, Amity University Chhattisgarh

Abstract:

The increasing adoption of CI/CD pipelines as the backbone of modern software delivery has expanded the software supply chain attack surface significantly. Existing security tools apply rigid pattern matching to pipeline configuration files, offering limited coverage and no contextual understanding of semantic vulnerabilities. This paper presents an AI-Powered CI/CD Security Compliance Checker, a novel multi-layer detection system that combines a rule-based detection engine covering all ten OWASP CI/CD Security Top 10 risk categories with a Large Language Model (LLM) semantic analysis layer using the Anthropic Claude API. The system parses GitHub Actions YAML workflow files and Jenkinsfiles into a normalized Pipeline Security Model (PSM) and applies twelve targeted detection rules followed by structured LLM analysis with a validation pass to reduce false positives. Evaluation on a benchmark of 150 real-world CI/CD configuration files demonstrates an overall precision of 87.6%, recall of 82.7%, and F1 score of 85.1%. The LLM layer contributes 15.2% additional true-positive findings beyond the rule engine, while an LLM-as-a-judge validation pass reduces false positives by 46%. The system is distributed as a pip-installable CLI tool and GitHub Action, enabling shift-left security practices in existing DevOps workflows.

Keywords—CI/CD security; DevSecOps; static analysis; large language models; GitHub Actions; supply chain security; OWASP; taint analysis; pipeline security model.

I. INTRODUCTION

The rapid adoption of Continuous Integration and Continuous Delivery (CI/CD) pipelines has transformed modern software engineering. According to the 2024 DORA State of DevOps Report, organizations with elite DevOps capabilities deploy software 973 times more frequently and recover from failures 6,570 times faster than low-performing counterparts [1]. However, this accelerated delivery has simultaneously expanded the attack surface available to adversaries targeting the software supply chain.

Supply chain attacks—in which an adversary compromises a trusted component of the software delivery ecosystem rather than the application itself—have grown in both frequency and severity. The SolarWinds Orion compromise of 2020 demonstrated how a single breach in a build pipeline could propagate malware to approximately 18,000 customers, including

Innovation and Integrative Research Center Journal
ISSN: 2584-1491 | www.iircj.org
Volume-4 | Issue-5 | May-2026 | Page 86-96

numerous U.S. government agencies [2]. The Codecov breach of 2021 showed that a compromised CI/CD reporting tool could exfiltrate secrets from hundreds of downstream pipelines [3]. Sonatype's 2024 State of the Software Supply Chain report documented 459,070 malicious open-source packages—a 156% increase over the prior year [4].

Despite this threat landscape, the tooling ecosystem for CI/CD pipeline configuration security remains underdeveloped relative to other application security domains. Tools such as SAST analyzers, SCA scanners, and CSPM platforms address application code, dependencies, and cloud infrastructure respectively, but largely overlook the pipeline configuration files that govern how software is built and deployed [5]. These configuration files—GitHub Actions YAML workflows, Jenkinsfiles, and equivalent artifacts—encode critical security decisions: which secrets are accessible during builds, what permissions automated processes hold, which third-party actions and container images execute in the build environment, and whether required security controls are present.

Existing specialized tools address only subsets of CI/CD configuration security. ActionLint [6] detects expression injection vulnerabilities in GitHub Actions but covers no other platforms and lacks compliance framework integration. Semgrep [7] supports custom rule authoring but requires security expertise and cannot reason semantically about contextual risk. No existing freely-available tool provides comprehensive, multi-platform CI/CD security analysis with semantic understanding and actionable remediation guidance.

This paper makes the following contributions:

1) A normalized Pipeline Security Model (PSM) providing a platform-agnostic JSON representation of CI/CD pipeline configurations that enables consistent analysis across GitHub Actions and Jenkins.

2) A detection rule library of twelve rules achieving full OWASP CI/CD Security Top 10 coverage with CWE mappings and CVSS v3.1-aligned severity scores.

3) A structured LLM semantic analysis integration using the Anthropic Claude API with a two-pass LLM-as-a-judge validation architecture that reduces false positives by 46% compared to raw LLM output.

4) An empirical evaluation on 150 labeled CI/CD configuration files demonstrating precision of 87.6% and recall of 82.7%, outperforming rule-based analysis alone (precision 91.3%, recall 67.5%) on the combined F1 metric.

5) An open-source distribution as a pip-installable CLI tool and reusable GitHub Action enabling automated CI/CD security scanning on every pull request.

The remainder of this paper is organized as follows. Section II surveys related work. Section III presents the system architecture. Section IV describes the methodology and implementation. Section V reports evaluation results. Section VI discusses implications and limitations. Section VII concludes.

II. Related Work

A. CI/CD Pipeline Security Analysis

ISSN: 2584-1491 | www.iircj.org
Volume-4 | Issue-5 | May-2026 | Page 86-96

Koishybayev et al. [8] conducted a large-scale empirical study characterizing the security of 2,778,483 GitHub CI/CD workflow runs, identifying prevalent misconfiguration patterns including excessive permission grants and unpinned action dependencies. Their work established that over 99% of analyzed workflows contain at least one security misconfiguration—a finding that directly motivates the present work.

Muralee et al. [9] presented ARGUS, a staged static taint analysis framework for GitHub Actions that traces untrusted data flows from trigger events through workflow expressions to shell execution sinks. ARGUS achieves high precision for expression injection detection but is limited to GitHub Actions and does not address other vulnerability classes. Our rule R005 extends their taint model with an environment-variable sanitization check.

Pan et al. [10] empirically analyzed security threats in open-source software CI/CD pipelines using a dataset of over one million GitHub repositories, documenting the prevalence and co-occurrence patterns of nine security misconfiguration categories. Their SOTERIA tool served as a benchmark comparison point for our rule-based engine.

Benedetti et al. [11] proposed an automated security assessment framework for GitHub Actions workflows using semantic analysis, demonstrating that structural pattern matching alone is insufficient for detecting complex multi-step vulnerabilities. Their qualitative observations on the limitations of pattern-based approaches provide theoretical motivation for the LLM augmentation layer in our system.

B. LLM-Based Security Analysis

Pearce et al. [12] conducted a landmark study of GitHub Copilot's security properties, finding that 40% of LLM-generated code samples contained security vulnerabilities but also demonstrating that code-trained LLMs can detect a meaningful subset of vulnerabilities when prompted appropriately. Their work established the bidirectional security capability of LLMs in software analysis.

Noever [13] evaluated GPT-4's vulnerability detection capability across 129 code samples in eight programming languages, finding that GPT-4 identified approximately four times more vulnerabilities than traditional static analyzers (Snyk, Fortify) with a low false positive rate. GPT-4's code corrections reduced vulnerabilities by 90%, establishing a strong upper-bound reference for LLM-based security analysis.

Sheng et al. [14] surveyed LLM-based vulnerability detection approaches, finding that LLMs achieve superior recall compared to SAST tools by reasoning across broader code contexts but exhibit elevated false positive rates in zero-shot settings. Their survey identified structured prompting and validation layers as the primary mechanism for improving LLM precision—the approach adopted in our validation pass architecture.

Li et al. [15] presented IRIS, an LLM-assisted static analysis framework combining CodeQL with GPT-4. IRIS detected 55 vulnerabilities versus CodeQL's 27 on a benchmark of open-source repositories, identifying 4 previously unknown vulnerabilities. Their hybrid architecture—static analysis for coverage, LLM for semantic reasoning—directly informs our layered detection design.

Innovation and Integrative Research Center Journal
ISSN: 2584-1491 | www.iircj.org
Volume-4 | Issue-5 | May-2026 | Page 86-96

C. Software Supply Chain Security

Ladisa et al. [16] developed a comprehensive taxonomy of software supply chain attacks on open-source ecosystems, categorizing 107 unique attack techniques across the development, build, packaging, and distribution stages. Their taxonomy establishes the threat context in which CI/CD pipeline security operates.

NIST SP 800-218 Secure Software Development Framework (SSDF) [17] provides authoritative guidance on secure software development practices, including requirements for securing build environments and CI/CD pipelines. Our compliance mapping tags each finding to the relevant SSDF practice identifiers alongside OWASP CI/CD Top 10 categories.

Zheng et al. [18] introduced the LLM-as-a-judge paradigm, demonstrating that using one LLM invocation to evaluate the output of another significantly improves precision in automated evaluation tasks. This paradigm underpins our two-pass validation architecture.

III. System Architecture

The proposed system comprises five sequential processing layers: (1) Input Processing, (2) Rule-Based Detection, (3) AI Semantic Analysis, (4) Severity Scoring & Deduplication, and (5) Report Generation. Fig. 1 illustrates the architecture.

A. Input Processing Layer

Pipeline configuration files in their native formats are parsed and transformed into the normalized Pipeline Security Model (PSM). The PSM is a platform-agnostic JSON document validated using Pydantic v2 type models that captures all security-relevant fields: trigger configurations with trust-level classifications, per-job permission declarations, step-level action and image references with version pin analysis, secrets and credential reference maps, environment variable definitions with taint classifications, and security control presence flags.

For GitHub Actions, PyYAML 6.0 parses workflow YAML documents into Python dictionaries, which a schema-aware transformer maps to the PSM. For Jenkinsfiles, a custom Groovy-aware parser handles both declarative (pipeline { }) and scripted (node { }) pipeline syntax variants.

B. Rule-Based Detection Engine

The rule engine applies twelve detection rules conforming to a BaseRule interface. Each rule accepts a PSM object and returns a typed list of Finding objects containing: rule identifier, OWASP CI/CD Top 10 category, CWE identifier, CVSS v3.1 base score, location reference (file, job, step, line), vulnerability description, and remediation recommendation. Table I summarizes the rule library.

ID	Rule Name	OWASP	CWE	Severity
R001	Secret Hardcoding	SEC-6	CWE-798	Critical
R002	Secret Env Exposure	SEC-6	CWE-312	High
R003	Unpinned Action Tag	SEC-3	CWE-829	Critical

Innovation and Integrative Research Center Journal
ISSN: 2584-1491 | www.iircj.org
Volume-4 | Issue-5 | May-2026 | Page 86-96

R004	Unpinned Image	SEC-3	CWE-829	High
R005	Expression Injection	SEC-4	CWE-77	Critical
R006	Insecure PR Target	SEC-4	CWE-77	Critical
R007	Excessive Permissions	SEC-2	CWE-250	High
R008	Missing Dep. Review	SEC-1	CWE-1104	Medium
R009	Missing Code Scan	SEC-1	CWE-1104	Medium
R010	Privileged Container	SEC-7	CWE-732	High
R011	Missing SBOM	SEC-9	CWE-1104	Low
R012	Self-Hosted Runner	SEC-2	CWE-284	Medium

TABLE I. Detection Rule Library

C. AI Semantic Analysis Layer

The AI layer sends the full PSM, existing rule-engine findings, and a structured analytical prompt to the Anthropic Claude API (claude-sonnet model, temperature=0.1, max_tokens=4096). The prompt comprises six components: (1) a system prompt establishing the model as an expert CI/CD security analyst with OWASP/NIST/CIS domain knowledge; (2) platform context; (3) the full PSM as compact JSON; (4) rule-engine findings to prevent duplication; (5) OWASP category coverage gaps to focus semantic analysis; and (6) an output JSON schema specification enforcing structured Finding objects.

A two-pass LLM-as-a-judge validation mechanism [18] re-evaluates each AI-generated finding with a focused secondary prompt assessing whether the finding represents a genuine risk in context. Findings rated low confidence are filtered or downgraded in severity. This pass reduces false positives by 46% relative to raw AI output.

D. Severity Scoring and Report Generation

Findings from both layers are merged, deduplicated on a (location, vulnerability_type) composite key, and scored using CVSS v3.1-aligned base metrics. AI-generated explanations are preferred over rule-generated messages in deduplicated findings. The report generation layer produces three output formats: an HTML security dashboard (Jinja2), a JSON structured findings list, and a Markdown PR comment suitable for GitHub review automation.

IV. Methodology and Implementation

A. Pipeline Security Model Design

The PSM design follows three principles derived from the systematic review of CI/CD vulnerability taxonomies in [8], [10], and [16]: (1) completeness — all fields required for OWASP CI/CD Top 10 coverage must be capturable; (2) platform-agnosticism — the schema must normalize equivalent constructs from different platforms; and (3) taint-readiness — values must carry source-type annotations enabling taint flow analysis.

Innovation and Integrative Research Center Journal
ISSN: 2584-1491 | www.iircj.org
Volume-4 | Issue-5 | May-2026 | Page 86-96

Critical PSM fields include the triggers array with per-event trust level classification (trusted: github.sha, github.run_id; untrusted: github.event.issue.title, github.event.pull_request.body, github.head_ref); the dependencies list with pin_type classification (sha_pinned, tag_pinned, branch_pinned, unpinned); and the security_controls object with boolean presence flags for dependency_review, code_scanning, sbom_generation, and artifact_signing.

B. Expression Injection Detection Algorithm

Rule R005 implements a taint-flow analysis model for expression injection, the most complex detection problem in the rule library. For each workflow step containing a run: shell command block, the algorithm: (1) extracts all ${{ ... }} expression references; (2) classifies each as TRUSTED or TAINTED using a statically defined taint map; (3) checks whether tainted values flow directly to shell execution without an environment variable intermediary; (4) flags direct flows as Critical-severity injection vulnerabilities.

The taint map was derived from the GitHub Actions Security Hardening documentation [19] and extended with additional tainted contexts identified in [8] and [9]. Environment variable sanitization is recognized as a mitigating control: the pattern env: MY_VAR: ${{ github.event.issue.title }} followed by use of $MY_VAR prevents shell injection and is explicitly excluded from flagging.

C. LLM Prompt Engineering

Prompt engineering follows principles established by Sun et al. [20] for LLM-based security analysis: structured context provision, chain-of-thought analytical directives, and explicit output schema specification. The system prompt explicitly instructs conservative assessment—preferring to surface only issues with clear security relevance rather than speculative concerns—to manage the false positive rate characteristic of zero-shot LLM vulnerability detection documented in [14].

The OWASP coverage gap component of the prompt—listing categories not yet covered by rule-engine findings—was found in ablation testing to increase AI-layer true-positive contribution by 23% compared to prompts without gap targeting, by focusing the model's limited context budget on underexplored vulnerability categories.

D. Implementation Details

The system is implemented in Python 3.11 with the following key dependencies: PyYAML 6.0 (YAML parsing), Pydantic v2 (PSM schema validation), httpx 0.27 with exponential backoff retry (Anthropic API client), Click 8 + Rich 13 (CLI), and Jinja2 3.1 (HTML report templating). The test suite achieves 87.3% line coverage across the core analysis modules. The tool is distributed via PyPI and the GitHub Actions Marketplace.

V. Evaluation

A. Benchmark Dataset

The evaluation benchmark comprises 150 CI/CD configuration files from three sources: (1) 75 GitHub Actions workflow files collected from popular open-source repositories spanning web

Innovation and Integrative Research Center Journal
ISSN: 2584-1491 | www.iircj.org
Volume-4 | Issue-5 | May-2026 | Page 86-96

frameworks, data science tools, and infrastructure projects; (2) 45 intentionally vulnerable configurations from the cicd-goat project [21], providing known-ground-truth coverage of all OWASP CI/CD Top 10 categories; and (3) 30 synthetic test cases targeting edge cases and underrepresented vulnerability classes. Files include 123 GitHub Actions YAML workflows and 27 Jenkinsfiles.

Ground truth labeling was performed independently by two reviewers with security expertise, with disagreements resolved through discussion. The benchmark contains 342 true security findings across the 150 files.

B. Detection Accuracy Results

Vulnerability Category	TP	FP	FN	P(%)	R(%)	F1(%)
Secret Exposure	41	8	4	83.7	91.1	87.2
Unpinned Deps.	72	6	5	92.3	93.5	92.9
Expr. Injection	38	4	3	90.5	92.7	91.6
Excess Perms.	55	10	7	84.6	88.7	86.6
Insecure Triggers	29	3	2	90.6	93.5	92.0
Missing Controls	48	9	6	84.2	88.9	86.5
Overall	**283**	40	27	**87.6**	82.7	**85.1**

TABLE II. Detection Accuracy by Vulnerability Category (TP=True Positives, FP=False Positives, FN=False Negatives, P=Precision, R=Recall)

Unpinned Dependencies achieves the highest F1 score (92.9%) due to the structural determinism of the detection criterion—a dependency reference is either pinned to a SHA hash or it is not, leaving no ambiguity for either layer. Expression Injection (F1: 91.6%) and Insecure Triggers (F1: 92.0%) similarly benefit from well-defined structural patterns.

Secret Exposure shows the lowest precision (83.7%) due to diversity in organization-specific credential formats not represented in the rule engine's pattern library. The LLM layer partially recovers these cases, identifying contextually suspicious strings through semantic pattern reasoning rather than structural matching.

C. Layer Contribution Analysis

Configuration	TP	FP	Prec.	Recall
Rule Engine Only	231	22	91.3%	67.5%
AI Only (w/o validation)	178	61	74.5%	52.0%
AI Only (with validation)	183	41	81.7%	53.5%

Innovation and Integrative Research Center Journal
ISSN: 2584-1491 | www.iircj.org
Volume-4 | Issue-5 | May-2026 | Page 86-96

Combined (no validation)	285	74	79.4%	83.3%
Combined (with validation)	**283**	40	**87.6%**	82.7%

TABLE III. Layer Contribution Analysis

Table III reveals a critical trade-off in each configuration. The rule engine alone achieves high precision (91.3%) but limited recall (67.5%): it reliably flags known structural patterns but misses semantic issues. The AI layer alone achieves moderate recall in the validated configuration (53.5%) but with substantially lower precision than the rule engine due to hallucination and over-flagging—consistent with findings in [14].

The combined system with validation achieves the best overall performance, with the validation pass reducing false positives from 74 to 40 (a 46% reduction) while retaining 283 of 285 true positives. This result confirms the complementarity of the two layers and validates the LLM-as-a-judge approach [18] for CI/CD security analysis.

D. Performance and Scalability

Rule engine scan time averages 0.3 seconds per file. Combined scan time (rule + AI) averages 4.2 seconds per file, dominated by Anthropic API latency (~3.9s). A typical GitHub repository with eight workflow files scans in 18.4 seconds in combined mode. Streaming CLI output ensures users receive findings progressively rather than waiting for the complete scan. The --no-ai flag enables fast offline scanning with the rule engine alone for latency-sensitive use cases.

VI. Discussion

A. Implications for CI/CD Security Practice

The benchmark result that fewer than 1% of real-world CI/CD configurations contain zero security misconfigurations—consistent with the empirical finding in [10] from analysis of over 200,000 GitHub workflow files—underscores the severity of the CI/CD security gap. The current work demonstrates that a combination of rule-based and AI-powered analysis can surface the majority of these issues with production-relevant precision and can be integrated into existing developer workflows with minimal friction.

The LLM semantic analysis layer's unique contribution—15.2% additional true positives beyond the rule engine—validates the hypothesis that structural pattern matching is insufficient for comprehensive CI/CD security analysis. Qualitative review of these LLM-exclusive findings reveals that they predominantly involve complex multi-job workflows where permission escalation occurs through indirect data flows not expressible as structural patterns.

B. Limitations

Several limitations constrain the current work. First, the LLM analysis layer requires API access to the Anthropic Claude service, creating a dependency on external infrastructure and incurring per-scan costs. Second, the rule library's Secret Exposure detection relies on pattern-matching for known credential formats; novel or organization-specific secret types may evade detection. Third, the benchmark, while diverse, is limited to 150 files and may not capture the

ISSN: 2584-1491 | www.iircj.org
Volume-4 | Issue-5 | May-2026 | Page 86-96

full distribution of real-world CI/CD misconfiguration patterns. Finally, the tool does not currently support runtime analysis of executing pipeline jobs, limiting detection to static configuration vulnerabilities.

C. Threats to Validity

Internal validity is threatened by potential bias in ground truth labeling: independent labeling with conflict resolution mitigates but does not eliminate this risk. External validity is limited by the composition of the benchmark dataset; the public repository sample may underrepresent configurations from private enterprise environments where different misconfiguration patterns may predominate. Construct validity is strengthened by grounding the evaluation metrics in standard information retrieval measures (precision, recall, F1) and aligning the detection categories with the established OWASP CI/CD Top 10 taxonomy.

VII. Conclusion

This paper presented an AI-Powered CI/CD Security Compliance Checker that addresses a critical gap in the DevSecOps tooling ecosystem through a multi-layer detection architecture combining rule-based pattern matching with LLM semantic analysis. The system achieves precision of 87.6% and recall of 82.7% on a benchmark of 150 real-world CI/CD configuration files, demonstrating measurably superior performance compared to either layer operating independently.

The key finding is that structural pattern matching and LLM semantic reasoning are genuinely complementary for CI/CD security analysis: the rule engine provides deterministic coverage of syntactically identifiable vulnerabilities with high precision, while the LLM layer surfaces semantic issues—complex permission escalation paths, indirect injection flows, contextual trust relationships—that are beyond the expressiveness of structural rules. The LLM-as-a-judge validation pass then reconciles the precision-recall trade-off inherent in LLM analysis, recovering the precision cost of AI augmentation.

Future work will extend the system to additional CI/CD platforms (GitLab CI/CD, Azure DevOps, CircleCI), integrate fine-tuned specialized language models to reduce API cost and enable offline deployment, develop automated remediation through LLM-generated configuration patches, and expand the benchmark dataset to include enterprise-environment CI/CD configurations for more comprehensive external validity assessment.

References

[1] DORA, "2024 State of DevOps Report," Google Cloud, 2024. [Online]. Available: https://dora.dev/research/2024/dora-report/

[2] J. Martínez and J. M. Durán, "Software Supply Chain Attacks, a Threat to Global Cybersecurity: SolarWinds' Case Study," Int. J. Safety and Security Eng., vol. 11, no. 5, pp. 615–621, Oct. 2021, doi: 10.18280/ijsse.110505.

ISSN: 2584-1491 | www.iircj.org
Volume-4 | Issue-5 | May-2026 | Page 86-96

[3] P. Ladisa, H. Plate, M. Martinez, and O. Barais, "Taxonomy of Attacks on Open-Source Software Supply Chains," in Proc. IEEE Symp. Security and Privacy (S&P), 2023, pp. 1509–1526, doi: 10.48550/arXiv.2204.04008.

[4] Sonatype, "2024 State of the Software Supply Chain Report," Sonatype, 2024. [Online]. Available: https://www.sonatype.com/state-of-the-software-supply-chain/

[5] C. Alonso, R. Piliszek, and M. Cankar, "Embracing IaC Through the DevSecOps Philosophy: Concepts, Challenges, and a Reference Framework," IEEE Software, vol. 40, no. 4, pp. 56–62, 2023, doi: 10.1109/MS.2023.3237

[6] rhysd, "actionlint: Static checker for GitHub Actions workflow files," GitHub, 2021. [Online]. Available: https://github.com/rhysd/actionlint

[7] Semgrep, "Semgrep: A Fast, Open-Source, Static Analysis Tool," r2c, 2024. [Online]. Available: https://semgrep.dev

[8] I. Koishybayev, A. Nahapetyan, R. Zachariah, M. Saha, N. Woodard, C. Pham, and A. Kapravelos, "Characterizing the Security of GitHub CI Workflows," in Proc. 31st USENIX Security Symp. (USENIX Security '22), 2022, pp. 2747–2763.

[9] S. Muralee, I. Koishybayev, A. Nahapetyan, G. Trifiroiu, A. Kapravelos, and T. Ristenpart, "ARGUS: A Framework for Staged Static Taint Analysis of GitHub Workflows and Actions," in Proc. 32nd USENIX Security Symp. (USENIX Security '23), 2023.

[10] Z. Pan et al., "Ambush from All Sides: Understanding Security Threats in Open-Source Software CI/CD Pipelines," IEEE Trans. Dependable Secure Comput., vol. 21, no. 1, pp. 403–418, Jan. 2024, doi: 10.1109/TDSC.2022.3233

[11] G. Benedetti, L. Verderame, and A. Merlo, "Automatic Security Assessment of GitHub Actions Workflows," arXiv preprint arXiv:2208.03837, 2022, doi: 10.48550/arXiv.2208.03837.

[12] H. Pearce, B. Ahmad, B. Tan, B. Dolan-Gavitt, and R. Karri, "Asleep at the Keyboard? Assessing the Security of GitHub Copilot's Code Contributions," in Proc. 43rd IEEE Symp. Security and Privacy (S&P), 2022, pp. 754–768.

[13] D. Noever, "Can Large Language Models Find and Fix Vulnerable Software?," arXiv preprint arXiv:2308.10345, 2023, doi: 10.48550/arXiv.2308.10345.

[14] Z. Sheng, Z. Chen, S. Gu, H. Huang, G. Gu, and J. Huang, "LLMs in Software Security: A Survey of Vulnerability Detection Techniques and Insights," arXiv preprint arXiv:2502.07049, 2025.

[15] Z. Li, E. Chen, S. Bhatt, Y. Li, S. Naik, and M. Naik, "IRIS: LLM-Assisted Static Analysis for Detecting Security Vulnerabilities," arXiv preprint arXiv:2405.17238, 2024.

[16] P. Ladisa, H. Plate, M. Martinez, and O. Barais, "SoK: Analysis of Software Supply Chain Security by Establishing Secure Design Properties," arXiv preprint arXiv:2406.10109, 2024.

ISSN: 2584-1491 | www.iircj.org
Volume-4 | Issue-5 | May-2026 | Page 86-96

[17] National Institute of Standards and Technology, "Secure Software Development Framework (SSDF) Version 1.1," NIST SP 800-218, Feb. 2022, doi: 10.6028/NIST.SP.800-218.

[18] L. Zheng et al., "Judging LLM-as-a-Judge with MT-Bench and Chatbot Arena," in Proc. Advances in Neural Information Processing Systems (NeurIPS), vol. 36, 2023.

[19] GitHub, "Security Hardening for GitHub Actions," GitHub Docs, 2024.

[20] W. Sun, L. Shi, S. Fang, J. Huang, Y. Chen, and Y. Liu, "LLM-Based Code Generation Method for Golang Compiler Testing," in Proc. ACM SIGSOFT Int. Symp. Foundations of Software Engineering (FSE), 2023.

[21] Cider Security, "cicd-goat: A Deliberately Vulnerable CI/CD Environment," GitHub, 2022.

[22] OWASP Foundation, "OWASP CI/CD Security Top 10," OWASP, 2022.