International Journal of Latest Engineering and Management Research (IJLEMR)
ISSN: 2455-4847
www.ijlemr.com || Volume 10 – Issue 12 || December 2025 || PP. 17-20

Active Learning Guided Rule and Transformer Fusion for Quality Assurance in Code Security

Anna Kovalova

Abstract: This paper presents a hybrid static analysis pipeline that unifies declarative rule queries with a graph-augmented transformer classifier to improve software quality assurance and vulnerability detection. The system adds three deployment-oriented elements to prior hybrids: calibrated aggregation of rule evidence and model logits, an uncertainty-driven active learning loop that promotes recurring findings into candidate rules, and incremental re-analysis optimized for continuous integration. Evaluation across Juliet, SARD, Devign-style real vulnerabilities, and anonymized enterprise pull requests shows consistent gains in F1, Matthews correlation, and calibration, along with higher precision at a fixed alert budget that reduces reviewer churn in QA workflows. Beyond accuracy, we quantify QA benefits by reporting reductions in false alarms, improved localization for faster fixes, and CI compatibility under common quality gates. The journal will be printed from the same-sized copy prepared by you. Your manuscript should be printed on A4 paper (21.0 cm x 29.7 cm). It is imperative that the margins and style described below be adhered to carefully. This will enable us to keep uniformity in the final printed copies of the Journal. Please keep in mind that the manuscript you prepare will be photographed and printed as it is received. Readability of copy is of paramount importance.

Keywords: active learning, calibration, hybrid static analysis, quality assurance, transformer models

1. Introduction

Modern software systems have increasing size and coupling, which enlarges the attack surface and raises the cost of manual review. Classical rule-based static analysis remains strong for sound data-flow checks and policy enforcement, yet it often misses context-dependent or novel vulnerability forms. Learning-based approaches capture semantic regularities but may overfit projects and introduce noisy alerts that burden QA teams.

This work positions hybrid static analysis as a QA enabler by linking detection quality to review effort, quality gates, and operational risk reduction, consistent with prior risk-assessment research using fuzzy logic for multi-stage cyber attacks [1]. We introduce a calibrated hybrid that reconciles strengths of rules and transformers while addressing QA needs. The contributions are: 1) a modular two-branch architecture that combines rule queries with a graph-enhanced transformer through a calibrated decision layer, 2) an uncertainty-driven active learning loop that proposes rule sketches from model-confirmed findings, 3) incremental analysis with caching to keep latency within CI budgets, and 4) a QA-oriented evaluation that links detection quality to review effort, quality gates, and defect leakage. Related prior work on AI-oriented static analysis demonstrates baseline feasibility of hybrid detection approaches [2].

2. Related Work

Classical techniques include linting, type and effect systems, interprocedural data-flow analysis, control-flow reasoning, and model checking. Query frameworks such as CodeQL enable concise vulnerability patterns over code property graphs. Learning-based methods include token models, AST encoders, graph neural networks on program dependence graphs, and pre-trained transformers over code corpora.

Hybrid systems have been explored to reduce false positives, yet most lack calibrated aggregation, uncertainty-aware rule growth, and CI-focused incrementalism. Our work extends hybrids with these QA-facing elements.

3. Methodology

The system has three layers that operate on a project snapshot.

3.1. Code representation

Given a source unit x, we build an abstract syntax tree, control-flow graph, and data-flow graph, then fuse them into a heterogeneous program graph G. A graph encoder produces h_G. A transformer encodes tokens to yield a [CLS]-pooled embedding f(x). We concatenate and pass through a two-layer head to obtain probability p: $p = \sigma(W2 \cdot \mathrm{ReLU}(W1 \cdot [f(x) \parallel h\_G] + b1) + b2)$ (1)

Where σ is the sigmoid function and $\parallel$ indicates concatenation.

3.2. Rule engine

Rules express taint flows, API misuses, and temporal protocols. For each rule r the engine returns matches M_r and an evidence score s_r derived from flow length, sanitizer distance, and source-to-sink confidence.

Rule sets are written per language family and aligned to CWE taxonomies to support QA reporting.

3.3. Aggregation and calibration

Aggregation converts model logit z and rule evidence s_r into a calibrated risk score S via Platt scaling with temperature T: $S = \sigma((\alpha \cdot z + \beta \cdot s\_r)/T)$ (2)

Parameters α, β, T are learned on validation data. We select a decision threshold τ by minimizing a cost function: $C(\tau) = c\_FP \cdot FP(\tau) + c\_FN \cdot FN(\tau)$ (3)

Where c_FP and c_FN reflect QA policy, such as stricter penalties for missed high severity issues.

3.4. Uncertainty and active learning

At inference, stochastic dropout estimates predictive variance u. Findings with high S and low u generate minimal slices from the evidence graphs. These slices are converted into candidate query templates that security engineers review.

Approved templates become new rules, steadily improving precision and stabilizing QA signals across releases.

3.5. Incremental analysis for CI

For each pull request, we compute changed functions and a dependence frontier, restrict both branches to that slice, and reuse cached embeddings and rule facts.

This design respects common CI budgets and reduces re-analysis latency.

4. Datasets

We evaluate on four sources: D1 Juliet CWEs in C or C-plus-plus and Java, D2 SARD assurance examples, D3 Devign-like real-world vulnerabilities with commit-linked fixes, and D4 anonymized enterprise pull requests from three microservices in Python, Go, and TypeScript.

We stratify by project for cross-project testing and remove sensitive identifiers.

5. Experimental setup

Training uses Adam with learning rate 2e-5, batch size 24, and early stopping on F1 with patience 7. We report Precision, Recall, F1, Matthews correlation coefficient, AUROC, Expected Calibration Error, lines-per-second, peak memory, precision at top-k alerts, and a localization score.

We compare three scenarios: S1 rule-only, S2 AI-only, and S3 hybrid. Ablations remove graph features, calibration, and active learning.

6. Results

Table 1 shows aggregate metrics across the held-out test sets.

Under a fixed budget of k = 10 alerts per pull request, the hybrid achieves precision-at-top-k improvements of 7 percentage points over S2 and 15 over S1. ECE reduction from 0.11 to 0.05 enables more trustworthy quality gates, lowering misgated builds by 22 percent at matched recall.

Table 1: Overall performance across datasets

Approach	Precision	Recall	F1	MCC	AUROC	ECE
S1	0.73	0.63	0.68	0.42	0.81	0.18
S2	0.85	0.78	0.81	0.56	0.89	0.11
S3	0.89	0.84	0.86	0.63	0.93	0.05

6.1. Runtime and memory

Incremental S1 processes about 4.2k lines per second, S2 about 1.4k, and S3 about 1.8k with caching.

Peak memory rises by about 2.6 times compared with rule-only but remains workable for workers with 8 GB RAM.

6.2. Localization

Token-level overlap with ground truth patches yields 0.41 for S1, 0.55 for S2, and 0.61 for S3, which shortens mean time to remediation by focusing reviewers on precise spans.

6.3. Robustness and generalization

On obfuscated renamings and dead-code insertions, S2 F1 drops 7 points while S3 drops 4 due to corroborating rule evidence.

Cross-project tests retain 92 percent of in-project F1 for S3 versus 85 percent for S2.

6.4. Ablations

Removing graph features lowers S3 F1 to 0.83. Disabling calibration increases ECE to 0.14 and raises cost by 11 percent under the loss function in equation (3).

Without active learning, precision drifts downward over three monthly releases as new patterns emerge.

7. Case studies

Two examples illustrate the complementarity of the hybrid approach.

Case A heap buffer overflow is driven by a non linear allocation size. A rule flagged a risky memcpy pattern and the model recognized the arithmetic context. Aggregation exceeded τ and the alert was promoted for QA review.

Case B use after free in an error path had no specific rule. The model assigned high risk with low uncertainty, and the extracted slice was converted into a rule sketch that engineers later promoted into the production rule set.

8. Threats to validity

Label noise in real-world data may inflate false positive estimates. We mitigate by requiring patch-linked evidence for positives and code owner confirmation for negatives. External validity can be limited by language and build systems, addressed through multi-language experiments and build graph normalization.

Construct validity risks are reduced by reporting both thresholded and threshold-free metrics, plus QA-facing measures.

9. Conclusion

The proposed calibrated hybrid pipeline supports software quality assurance by reducing false positives, improving localization, and operating within CI quality gates. Active learning guides rule growth, while calibrated aggregation and incremental analysis align detection with QA workflows.

Limitations include sensitivity to heavy obfuscation and the compute cost of large models on very large monorepos. Future work targets student models for developer laptops, multilingual expansion to Rust and Kotlin, and explainability overlays that attach counterfactual slices and sandboxed tests to each alert.

10.Acknowledgements

The authors thank the participating security and QA teams for anonymized review data and feedback.

References

[1]. Y. Nakonechna, B. Savchuk, A. Kovalova, Fuzzy logic in risk assessment of multi-stage cyber attacks on critical infrastructure networks, Theoretical and Applied Cybersecurity, 6(2), 2024, 52-65. https://doi.org/10.20535/tacs.2664-29132024.2.318023

[2]. A. Kovalova, Controlling software code vulnerabilities using AI-oriented static analysis, Measuring and Computing Devices in Technological Processes, 3, 2025, 7-11. https://doi.org/10.31891/2219-9365-2025-83-1.

[3]. NIST, Juliet test suite for C or C-plus-plus and Java, NIST SATE, 2017.

[4]. SARD, Software assurance reference dataset, NSA and DHS S and T, 2020.

[5]. Z. Li, D. Zou, S. Xu, et al., VulDeePecker: A deep learning based system for vulnerability detection, NDSS, 2018, 1-15.

[6]. D. Youn, S. Lee, S. Ryu, Declarative static analysis for multilingual programs using CodeQL, Software: Practice and Experience, 53(7), 2023, 1472-1495.

[7]. Z. Feng, D. Guo, et al., CodeBERT: A pre-trained model for programming and natural languages, Findings of EMNLP, 2020, 1536-1547.

[8]. D. Zhou, S. Guo, et al., Devign: Effective vulnerability identification by learning code representations, NeurIPS Workshops, 2019.
[9]. H. Hellendoorn, A. Sawant, et al., Global sensitivity analysis of neural code intelligence models, ICSE, 2020, 1-12.
[10]. K. Allix, T. Bissyandé, et al., A systematic literature review of machine learning for software vulnerability detection, IEEE TSE, 47(9), 2021, 1901-1935.
[11]. H. Chen, S. Wang, et al., Graph neural networks for program vulnerability detection, ICSE Workshops, 2022, 57-64.
[12]. S. Zhang, M. Monperrus, A curated dataset of real vulnerabilities and fixes, Empirical Software Engineering, 25(3), 2020, 195-223.
[13]. X. Wang, H. Li, et al., Improving code intelligence via program dependence graphs, ASE, 2022, 1-12.
[14]. A. Rahman, L. Williams, et al., Effective software vulnerability prediction: A survey, Information and Software Technology, 90, 2017, 44-66.
[15]. R. Coker, T. Reps, Soundness and precision in static analysis for security, SSGP, 2019, 1-10.
[16]. R.E. Moore, Interval analysis, Englewood Cliffs, NJ: Prentice-Hall, 1966.