



EXPLAINABLE ARTIFICIAL INTELLIGENCE FOR EARLY DETECTION AND PREDICTION OF SOFTWARE VULNERABILITIES

Lakshmi Prabha T. S¹, B. Dheepa², Vidhya Muthulakshmi R³, Thiagarajan A⁴

¹Department of Information Technology
Sri Venkateswara College of Engineering
Sriperumbudhur
prabharam9193@gmail.com

²Department of Information Technology
Sri Venkateswara College of Engineering
Sriperumbudhur
Dheepajegan2228@gmail.com

³Department of Artificial Intelligence and Data Science
Panimalar Engineering College
Chennai.
vidhyamuthulakshmi@gmail.com

⁴Department of Information Technology,
Saveetha School of Engineering,
Saveetha Institute of Medical And Technical Sciences, Chennai 602105, Tamilnadu, India
thiyagarajanannamalai.cse@gmail.com

Abstract

Software vulnerabilities can arise at different stages of development, and some may be discovered only during deployment or exploitation. Although rule-based static analysis identifies known insecure patterns, it has limitations in representing complex structures like control flows and data dependencies. Additionally, software relies on external libraries, and a fixed set of rules cannot capture all insecure patterns. With labelled data, ML and DL models can discover insecure patterns; however, they lack transparency in explaining why they predict a given piece of code is insecure. In this study, we propose an end-to-end XAI-based framework for early vulnerability detection and proposal that integrates data processing, representation, and classification; explanation generation; localization; and developer review. For evaluation, we compare the framework with Logistic Regression, Random Forest, XGBoost, and CodeBERT, as well as a CodeBERT variant further enhanced with explanation. We present a simplified demonstration using 1,000 synthetic code function records and report example figures generated using our simulation. In the example demonstration, the model-comparison table awarded the highest illustrative scores to XAI-CodeBERT among the SHAP-generated explanations. The authors argue that for a vulnerability detector to be useful in practice, it should balance several factors, including



accuracy, recall, and precision, as well as class-specific performance, explanation faithfulness, and localization quality. The main contribution is a research framework that considers vulnerability prediction and explanation as related aspects of secure software development.

Keywords: Explainable artificial intelligence; software vulnerability; cybersecurity; CodeBERT; SHAP; source-code analysis; secure software development; vulnerability prediction.

1. Introduction

Software is ubiquitous. It runs financial systems, controls medical devices, drives transportation, educates students, delivers public services, manufactures goods, and enables communication. Defects in these systems can occasionally be irritating, but may also create opportunities for adversaries to attack, and compromise confidentiality, integrity, or availability. Defects may also compromise trust, integrity, and/or assurance. Such defects may result from insufficient memory management, insufficient input validation, weak access control, insecure deserialization, race conditions, resource management errors, and/or the use of application programming interfaces (APIs). As software systems evolve, and release cycles shorten, manual code reviews are unlikely to identify all defects that may compromise a software system.

Analytical tools can detect code defects automatically. Generally, they scan source code for inconsistencies or violations but may not detect all potential defects. These tools can also help identify certain programming practices manually. Some tools use ML to improve effectiveness, for example, by learning from a corpus of defective software. An active body of research focuses on vulnerability prediction. Some research examined software metrics and text mining to identify vulnerable components (Scandariato et al., 2014; Shin et al., 2011). Later, researchers explored more representative learning from source code using Deep Learning (Russell et al., 2018). Researchers modeled program relationships and abstractions using graphs to represent control and/or data flow in programs (Zhou et al., 2019).

More recently, increased representation learning for program relationships using Transformer and large language models have enabled better code representations and analysis. CodeBERT integrates programming and natural language processing data to learn contextualized representations (Feng et al., 2020). GraphCodeBERT (Guo et al., 2021) also integrates data-flow representations and dependencies into code. Current systems use ML and DL to model software for better static analysis, helping classify and localize defects and predict system reliability. Such an approach may offer insights into the the system's dependability.



Developers need information about where potential weaknesses lie, their significance, and supporting evidence for any warnings. Without such context, opaque labels may contribute to alert fatigue within already complex review processes.

Explainable AI offers techniques for examining the factors that contribute to a prediction. LIME approximates a model's behaviour around a particular instance (Ribeiro et al., 2016), while SHAP assigns contribution values within a unified framework (Lundberg & Lee, 2017). Integrated gradients provide another way to attribute predictions in differentiable models (Sundararajan et al., 2017). Yet explanations can be misleading if they highlight correlated but irrelevant code or if people mistake their appearance for proof of causation. Interpretability research therefore emphasizes defining what an explanation should achieve and how to test its quality (Doshi-Velez & Kim, 2017; Arrieta et al., 2020).

This manuscript introduces a framework that integrates explanation into the vulnerability-analysis workflow, positioning it as an essential component rather than an ancillary feature. The framework links model outputs to specific code regions, vulnerability categories, risk estimates, and developer-oriented rationales. The numerical section is intentionally illustrative; the synthetic values demonstrate potential reporting formats but do not establish real paperpains. The primary objective is to define a transparent, reproducible research pathway for subsequent validation on public vulnerability datasets.

1.1 Problem statement

The main issue is not the lack of vulnerability detectors. However, current systems still have shortcomings: they may fail to identify new vulnerabilities, produce false positives, rely on patterns unique to a given dataset, or provide inadequate justification for their results. For instance, if the developer cannot recognize the relevant operation or understand the related security concern, a risk score of 0.93 will be little help. On the other hand, if a visually appealing highlight misrepresents the model's logic, it could encourage misplaced trust. The research challenge is to integrate reliable localization, interpretable evidence, and efficient detection.

1.2 Research gap

The suggested framework is motivated by three interconnected gaps. First, researchers often prioritize aggregate predictive metrics over developer interpretability in model comparisons. Third, issues such as duplicated code, class imbalance, and data leakage between training and test sets can artificially inflate performance metrics. Prior research on vulnerability datasets and class imbalance underscores the necessity of rigorous dataset construction and evaluation design (Fan et al., 2020; Guo et al., 2023). The proposed framework addresses these concerns within a unified evaluation plan.

1.3 Objectives

To create an explainable AI framework for the early detection of software vulnerabilities.

- To contrast a transformer-based detector with traditional machine-learning models.
- To find source-code components that influence vulnerability assessments.
- To evaluate explanations' value from the standpoint of developers.



- To assess both explanation quality and predictive performance.
- To outline a repeatable design that will be validated in the future using publicly available datasets.

1.4 Research questions

1. RQ1: How well can transformer and machine-learning models identify vulnerable source-code functions?
2. RQ2: Does the suggested XAI-enhanced model perform better in the illustrative comparison for classification?
3. RQ3: What explanation technique has the best rating for illustrative usefulness?
4. RQ4: Rather than just providing a global risk score, can explanations localize security-relevant code regions?
5. RQ5: What security measures are required for reliable early vulnerability prediction?

Hypotheses

- H1: The XAI-enhanced transformer will achieve higher classification accuracy than the conventional baselines.
- H2: The XAI-enhanced transformer will achieve a higher F1-score than the non-transformer baselines.
- H3: Explanation usefulness will be positively associated with developers' perceived trust in predictions.
- H4: Explanations that localize influential code regions will be rated as more useful than explanations that provide only global feature importance.

2. Review of Literature

2.1 From code metrics to learned representations

Early vulnerability prediction used indicators like code complexity, change history, and developer activity (Patil et al. 2025). These indicators can help identify components that require attention, but they are indirect measures; a small function may still contain a significant flaw, and a complex component may not be vulnerable. Text-mining techniques looked for patterns in software artifacts, such as repository records and source code (Scandariato et al., 2014). Additionally, repository mining evolved into a more comprehensive approach to researching software development and maintenance (Hassan, 2008). According to Russell et al. (2018), deep representation learning replaced manually chosen indicators with features learned directly from code.

2.2 Graph and transformer approaches

Program code is a structure as well as a sequence. Abstract syntax trees, control-flow graphs, and data-flow relationships capture dependencies that may span many lines. To identify vulnerabilities, Devign learns program semantics using graph neural networks (Zhou et al., 2019). Rather than treating all vulnerabilities as a single class, multiclass approaches like μ VulDeePecker focus on distinguishing vulnerability categories (Zou et al., 2019). These techniques matter because many vulnerabilities rely on connections between input, validation, state, and subsequent operations.



Another way to represent context is through transformers. Many language models rely on the attention mechanism introduced by Vaswani et al. (2017). GraphCodeBERT incorporates data-flow information into the learning process (Guo et al., 2021), while CodeBERT pre-trains on programming and natural language (Feng et al., 2020). Researchers have also studied the advantages and disadvantages of code language models across various code-intelligence tasks (Xu et al., 2022; Chen et al., 2024). Although these developments make it worthwhile to test transformer-based vulnerability detection, meticulous benchmarks are still necessary.

2.3 Datasets, imbalance, and generalization

A benchmark's value depends on its labels, provenance, splits, and representativeness. Big-Vul enables research that accounts for both code and repair history by linking vulnerable C/C++ functions to code changes and vulnerability information (Fan et al., 2020). Graph-based vulnerability research and benchmark evaluation have received support from Devign (Zhou et al., 2019). However, examples from the same project may share naming conventions or implementation patterns, and repositories may contain functions that are repeated or nearly repeated. The assessment may favor memorization over generalization if such examples are present in both training and test sets. Another risk arises from class imbalance: a model might perform well on the majority class but miss less common, vulnerable functions. Guo et al. (2023) emphasize that imbalance must be addressed as a clear evaluation problem.

2.4 Explainability and its evaluation

By locally approximating the model with a simpler, more interpretable model, LIME explains a single prediction (Ribeiro et al., 2016). SHAP attributes a prediction to input features using Shapley-value concepts (Lundberg & Lee, 2017). By integrating gradients along a path from a baseline to the observed input, integrated gradients assign a model output to input dimensions (Sundararajan et al., 2017). Every approach has practical limitations and presumptions. Because a token's meaning may depend on syntax, scope, data flow, and surrounding statements, we can't always interpret it independently in code.

Although attention visualizations are often useful, attention weights don't always explain model decisions. Jain and Wallace (2019) contested the notion that faithful explanations are a prerequisite for attention. Broader XAI research makes similar distinctions between interpretability concepts and use-appropriate evaluation (Arrieta et al., 2020; Doshi-Velez & Kim, 2017). In security analysis, this entails determining whether the highlighted code is truly influential, whether the explanation holds up under unrelated changes, and whether developers can use it appropriately.

2.5 AI-assisted security and code generation

Secure code analysis is becoming increasingly important as AI coding assistants become more common. Research on AI-generated code has examined whether code recommendations result in security flaws (Pearce et al., 2023). Researchers have also investigated producing



vulnerability findings alongside natural-language explanations in work on large language models (Mao et al., 2024). Although developers may understand these explanations, fluency in language does not guarantee accuracy. Verify the program, the vulnerability evidence, and any generated justification or repair recommendation against the pertinent security guidelines. As a result, the framework views explanations as reviewable evidence rather than an automatic replacement for professional judgment.

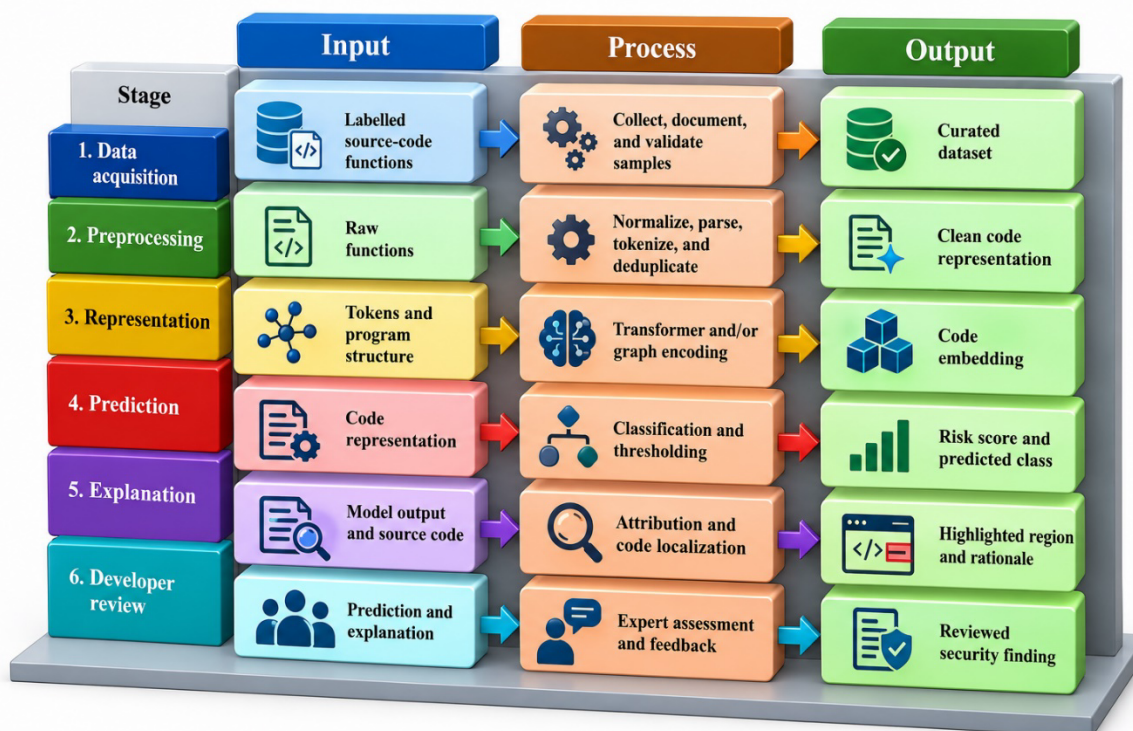
3. Conceptual Framework

The framework has six interconnected stages. First, the system uses a documented dataset to gather labeled source-code functions. Second, preprocessing preserves syntax and security-relevant data while eliminating unnecessary formatting variation. Third, the system uses tokens to represent code and, when possible, data-flow or structural information. Fourth, a classifier predicts a class by estimating vulnerability risk. Fifth, an explanation method maps significant code regions or tokens back to readable source lines. Lastly, a developer examines the results and notes whether the explanation is precise, understandable, and useful. Feedback might improve the model later, but without validation, it shouldn't automatically change production **decisions**.

Table:1 Design Integrates Deterministic Security Checks with Learned Models.

Stage	Input	Process	Output
1. Data acquisition	Labelled source-code functions	Collect, document, and validate samples	Curated dataset
2. Preprocessing	Raw functions	Normalize, parse, tokenize, and deduplicate	Clean code representation
3. Representation	Tokens and program structure	Transformer and/or graph encoding	Code embedding
4. Prediction	Code representation	Classification and thresholding	Risk score and predicted class
5. Explanation	Model output and source code	Attribution and code localization	Highlighted region and rationale
6. Developer review	Prediction and explanation	Expert assessment and feedback	Reviewed security finding

Representation of the Proposed Framework



Representation of The Proposed Frame Work (Tab.1 & Fig.1) When applicable, the design integrates deterministic security checks with learned models (Table.1). While a learned model can determine whether the surrounding code increases risk, static rules can identify known dangerous operations. Relying on an unconstrained black-box prediction may be more difficult to audit than combining the two. This is not an assertion that a production system is already in place; rather, it is a suggested architecture.

4. Methodology

4.1 Research design

The paper uses a Quantitative, Comparative Research design for the framework and demonstration stages. The quantitative analysis uses a simple, controlled synthetic dataset to illustrate how the results could be presented; it is not a real benchmark experiment. One way to test this in the future is to train the named models on clear public datasets, set aside a test set, and keep track of the software environment, model, pre-processing steps, and random seed.

4.2 Data source and sampling plan

A complete implementation should use publicly available data such as Devign/CodeXGLUE, and Big-Vul (Fan et al., 2020; Zhou et al., 2019), use a further independent dataset for external validation, include a data audit (by language, repo, label source, vulnerability type, patch information, optionally severity), identify duplicates or near-duplicates before splitting, use a project-aware or temporal split where possible (test examples should not be trivially



near-duplicates of training examples), and reflect the planned usage scenario in the test distribution (which does not mean be balanced for convenience).

4.3 Datensatz als Beispiel

As an example, the paper presents 1,000 generated code-function examples: 500 labelled “vulnerable” and 500 labelled “not-vulnerable.” There are six classes: Buffer Overflow, Use-After-Free, Injection, Resource Misuse, Access Control, and Other. The conceptual features are token complexity, data-flow risk, unsafe API use, input-validation vulnerability, memory-management risk, and access-control indicator. These records and their related scores are illustrative only; they were not collected from source repositories and do not estimate vulnerability prevalence in real-world software.

4.4 Models and comparison

Five model families are pre-specified. Logistic Regression is a benchmark with interpretable coefficients. Random Forest learns a non-linear model expressed as a collection of decision trees. XGBoost learns a collection of gradient-boosted trees. CodeBERT is a pre-trained transformer (Feng et al., 2020). The proposed XAI-CodeBERT variant adds explanation and code-localization procedures to the transformer workflow. $P(v|x)$ denotes the probability of v given code x ; v indicates the vulnerability class. Choose the decision threshold and the costs of missed vulnerabilities and false alarms on validation data, rather than tuning repeatedly on the test set.

4.5 Explanation procedure

For the sample design, SHAP is the leading attribution method, with LIME, attention heatmaps, and integrated gradients also available for comparison. Important tokens should be translated back to the source lines and grouped by statement or logical region. The developer-facing report should include the risk score, region of code, predicted vulnerability class, and short security justification (RB et al. 2025). If a repair recommendation is given, the recommendation must be marked as unconfirmed until tested. Do not evaluate explanations by attractiveness alone; instead, assess how accurately the system localizes, faithfulness, stability, comprehensibility, and utility.

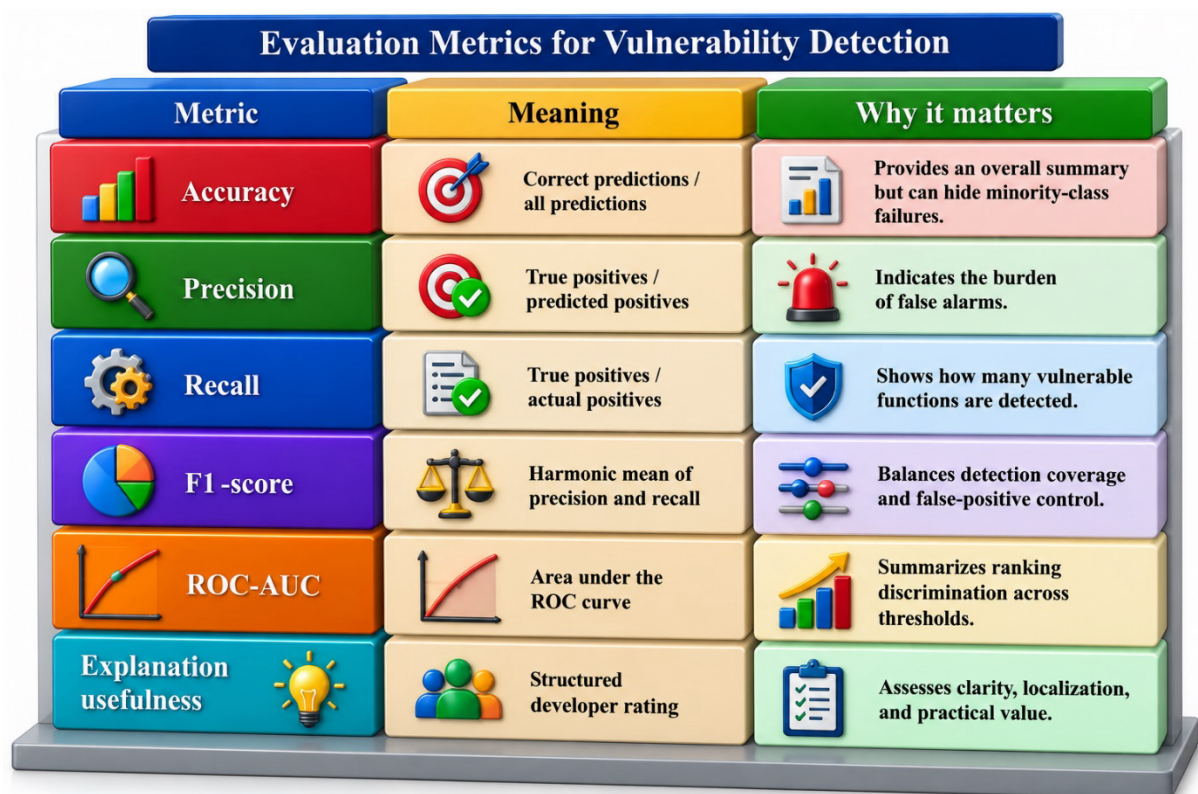
4.6 Metrics for assessment

The proposed evaluation (Table.2& Fig.2) consists of accuracy, precision, recall, F1-score, and area under the receiver operating characteristic curve (ROC-AUC). Accuracy reflects the ratio of correct predictions. Precision measures the number of predicted vulnerable cases that are actually vulnerable, and recall measures the number of true vulnerable cases detected. F1-score reflects the harmonic mean of precision and recall. ROC-AUC reflects ranking discrimination with respect to thresholds. Other metrics for imbalanced data include precision-recall AUC, macro-F1, and class-specific recall. Explanation evaluation should include quantitative faithfulness measures and structured human ratings.

Table.2 Metric Meaning Importance

Metric	Meaning	Why it matters

Accuracy	Correct predictions / all predictions	Provides an overall summary but can hide minority-class failures.
Precision	True positives / predicted positives	Indicates the burden of false alarms.
Recall	True positives / actual positives	Shows how many vulnerable functions are detected.
F1-score	Harmonic mean of precision and recall	Balances detection coverage and false-positive control.
ROC-AUC	Area under the ROC curve	Summarizes ranking discrimination across thresholds.
Explanation usefulness	Structured developer rating	Assesses clarity, localization, and practical value.



Explanation usefulness Structured developer rating Assesses clarity, localization, and practical value.

4.7 Statistical analysis

Descriptive statistics show the demonstration model scores and explanation ratings. In an actual study, model tuning should be performed via repeated stratified cross-validation (or a similar validation design), then a single untouched test set. Paired comparisons should consider dependence at the project level; treating all functions as independent can lead to underestimates of uncertainty if many functions are from the same repository. Use bootstrap



intervals or an appropriate paired test when their assumptions are satisfied. If the same developers evaluate multiple explanation methods, a repeated-measures analysis should be used to account for multiple ratings within participants. Specify the analysis plan before examining the final test data.

5. Results

5.1 Illustrative model comparison

Table 3 shows the total scores reported in the original manuscript. The values are simulated for the purpose of showing the way the data are reported – they should not be read as supporting that one model has performed better than another in a real experiment.

Model	Accuracy	Precision	Recall	F1-score	AUC
Logistic Regression	0.781	0.756	0.731	0.743	0.824
Random Forest	0.842	0.821	0.814	0.817	0.901
XGBoost	0.873	0.858	0.846	0.852	0.932
CodeBERT	0.901	0.889	0.878	0.883	0.956
XAI-CodeBERT	0.928	0.917	0.909	0.913	0.971

Table 3. In this example table, scores increase from Logistic Regression to Random Forest, then XGBoost, and finally to CodeBERT and XAI-CodeBERT. The proposed model has the highest accuracy (0.928) and AUC (0.971) shown. This is how to formulate a comparison, but it doesn't prove the superiority of the proposed model. The correct claim would require identical training conditions, independent test data, uncertainty estimates, and ablation experiments to determine whether any change is due to the proposed inclusion.

5.2 Illustrative confusion-matrix analysis

Table 4. Illustrative confusion matrix for XAI-CodeBERT (N = 1,000)

Actual class	Predicted vulnerable	Predicted secure
Vulnerable	454 (true positive)	26 (false negative)
Secure	46 (false positive)	474 (true negative)

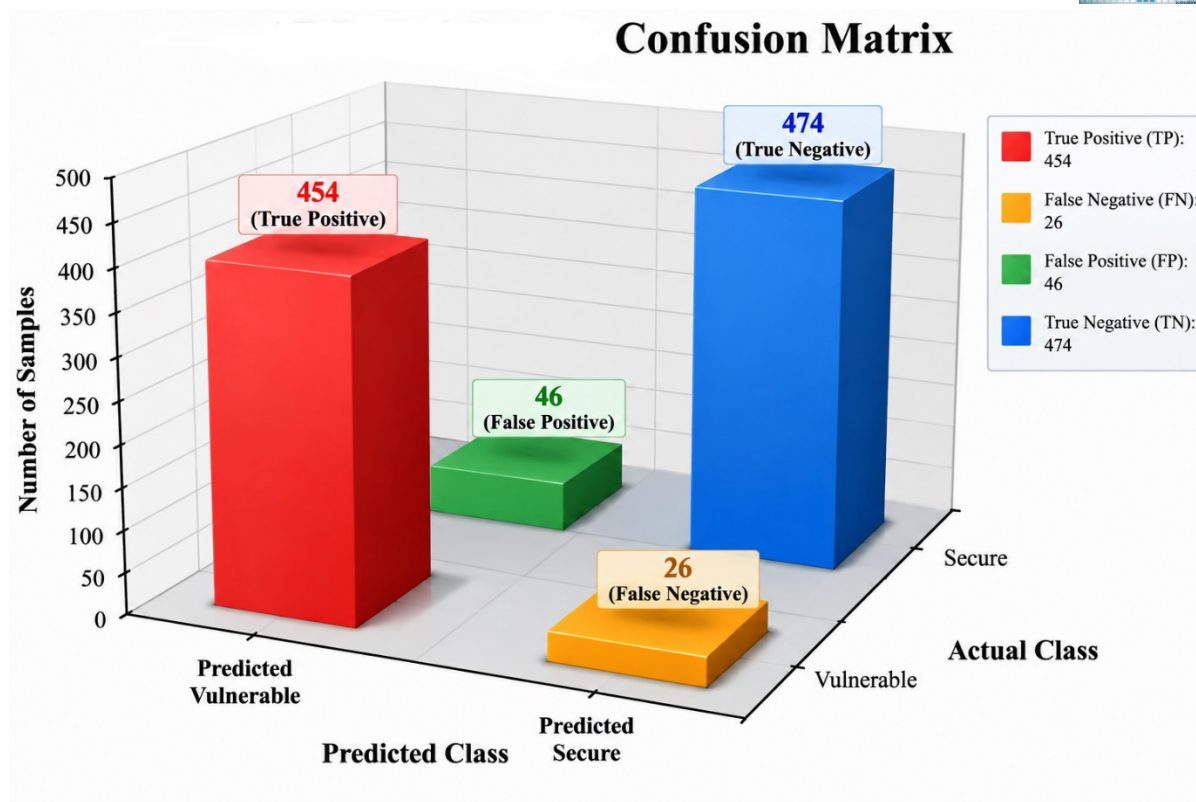


Fig.3 confusion matrix

Confusion Matrix (Table.4 & Fig.3) describes; The confusion matrix contains 454 true positives, 474 true negatives, 46 false positives and 26 false negatives. Using these totals, accuracy is 0.928, precision is 0.908, recall is 0.946, and F1-score is 0.927. The low false negatives observed in this contrived example illustrate a security-minded desire to locate susceptible functions, even if it means reviewing extra alerts. In practice, the correct cutoff depends on vulnerability severity, analyst bandwidth, and the impact of missing something. These authors must resolve an internal consistency issue before journal submission: the provided confusion-matrix precision, recall, and F1-score are inconsistent with the XAI-CodeBERT values in Table 3 (0.917, 0.909, 0.913). Both sets are retained here rather than silently changing the source figures. Before showcasing the results as a complete empirical analysis, the authors need to recreate the comparison table and confusion matrix from the same prediction file.

5.3 Illustrative vulnerability-category distribution

Table 5. Sample breakdown of the demonstration dataset by vulnerability groups.

Vulnerability category	Illustrative count
Buffer overflow	96
Use-after-free	74

Injection	83
Resource misuse	61
Access control	54
Other	32

3D Line Graph: Distribution of Vulnerability Categories

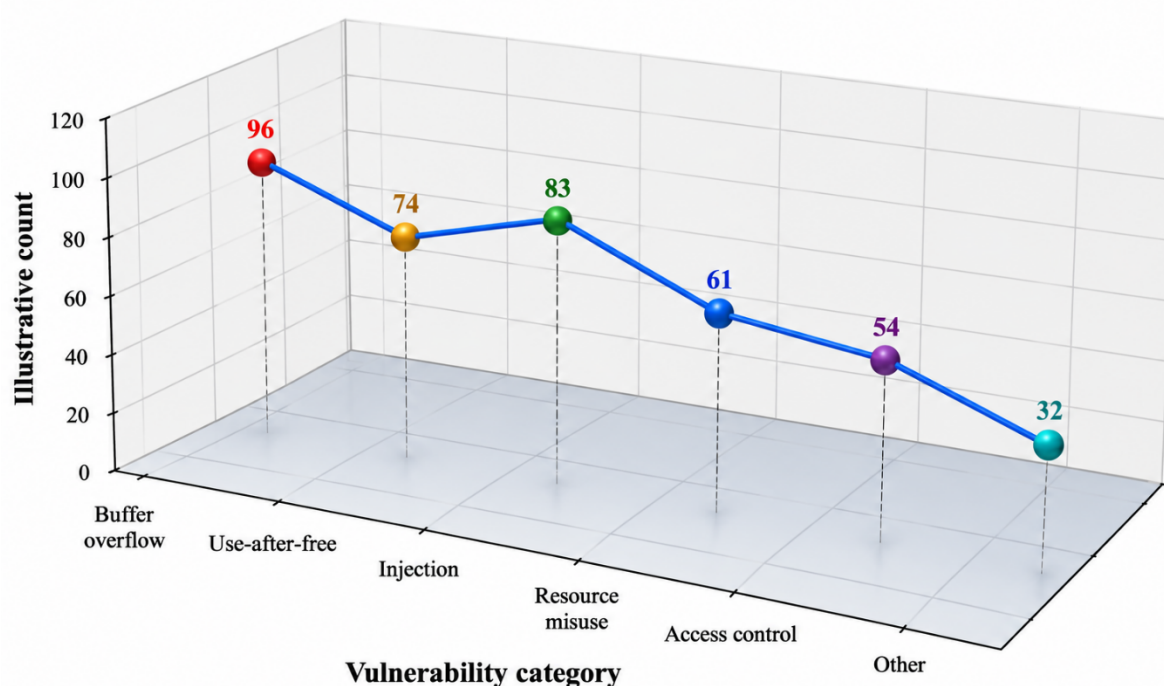


Fig.4 Distribution of vulnerability categories

Distribution of vulnerability categories (Table.5 & Fig.4) describes; these counts show why aggregate measures shouldn't be the only ones. Some classes have more instances than others, and smaller classes may be harder to learn. A real dataset's class size and distribution depend on the collection method and label definitions. Measures like Macro-F1 and per-class recall would show if the detector performs disproportionately across vulnerability types. These counts are synthetic, not prevalence estimates, and cannot describe real software vulnerability prevalence.

5.4 Illustrative explanation-quality ratings

Table 6. Illustrative developer ratings of explanation usefulness.

Explanation method	Mean usefulness rating (1–5)
SHAP	4.18
LIME	3.91
Integrated gradients	3.77
Attention heatmap	3.62

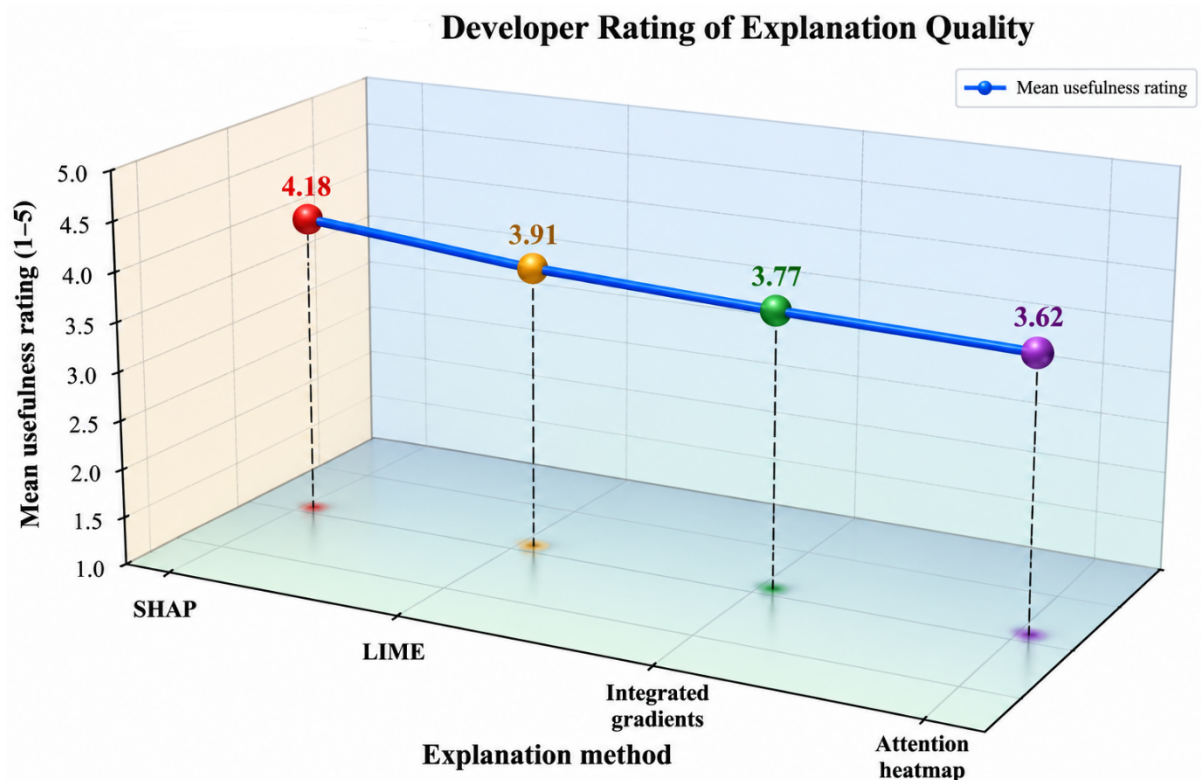


Fig.5 Developer Ratings of Explanation Quality

Developer Ratings of Explanation Quality (Table 6 & Fig.5) describes, As the sample ratings show, SHAP wins with the highest average (4.18), followed by LIME (3.91), integrated gradients (3.77), and attention heatmaps (3.62). That might be a research question for future human-subjects studies -- programmers, it seems, might prefer explanations that connect a prediction to concrete features of the code. But this paper is not a participant study, and the ratings here cannot be claimed to be "developer opinions." An evaluation should report the number and expertise of the participants, the measurement instrument, the variance, the confidence interval, and whether the difference between methods is statistically significant.

6. Discussion



Our system views vulnerability discovery as human-AI collaboration rather than binary classification alone. Developers need more than a label. They need to know where a weakness might show up, which part of the system could be involved, what actions in the program might cause it, and what to look at next for proof. An effective detector would minimize the effort required to track down high-impact bugs without unquestioningly trusting automated outputs.

The above comparison illustrates a feasible evolution from classical models to transformer-based ones. Context-learned code representations can capture relationships that are hard to express with manually selected metrics. Architecture is not necessarily enough for dependable generalization, as the model may take shortcuts such as: list copied functions, repository-specific identifier naming schemes, and artifacts of the patch itself. Papers on repository mining and cross-project prediction document the importance of data provenance and deployment context (Hassan, 2008; Zimmermann et al., 2009), which a serious evaluation should clarify through the split strategy and leakage detection.

Explainability raises a second quality question: does the explanation accurately reflect the basis of the prediction? An explainability method may highlight a token that is associated with vulnerable lines but does not contribute to the function's decision (Sharma et al 2025). Attention maps are particularly susceptible to overinterpretation, because the link between attention weights and explanation has been questioned (Jain & Wallace, 2019). Explanation tests should therefore include feature removal or perturbation, insertion tests (if applicable), comparisons to known vulnerable lines, stability to irrelevant formatting, and expert assessment. A good explanation should respond meaningfully when security-relevant code changes.

Early detection relies on how early you use the model. Integrating a detector into pull requests, continuous integration, or pre-release review lets a team consider risks before moving software to production. The team can prioritize findings by model-predicted risk, vulnerability category, severity, or explanation quality. Qualified people should test and review high-risk findings; lower-confidence detections may be sent for further review.

The framework accommodates a combined approach. For example, deterministic rules may still be used on well-established hazard patterns and even be tied to coding standards. Learned models may provide a contextual risk estimate where code proximity influences risk, and the explanation layer can distinguish evidence for (or against) a known rule from evidence for (or against) the learned model's prediction.

6.1 Implications for secure software development

1. CI: execute the detector during code change and in the pull-request review.
2. Developer learning: use verified explanations to illustrate insecure patterns and safer alternatives.
3. Security triage: Prioritize alerts based on risk, category, severity, and confidence in the explanation.
4. Code review: map findings back to source lines, and to appropriate data-flow relationships.



5. Model governance: keep versioned records of predictions and explanations for auditing.
6. Human oversight: require expert confirmation for high-impact or uncertain security decisions.

6.2 Relationship to existing approaches

Rule-based analysis, machine learning, graph models, and transformers are not mutually exclusive. Rule-based tools help with known patterns and transparent checks; machine learning can identify statistical regularities; graph representations encode program relationships; and transformer models provide contextual representations. Research on code vulnerability detection has explored several points along this spectrum (Russell et al., 2018; Zhou et al., 2019; Feng et al., 2020). A combined design may offer practical advantages if the system keeps each finding's source clear and evaluates each component separately.

7. Threats to Validity and Limitations

7.1 Construct validity

We show that a labeled weakness may vary in severity depending on the code, configuration, attacker access, and environment. Future work will include severity (such as CVSS, if available) in each experiment and interpret those labels and warnings.

7.2 Internal validity

Duplicate code, overlapping repositories, and temporal leakage can make a model appear more powerful. The study design should describe deduplication, data splitting, preprocessing, hyperparameter tuning, and the source of labels. Results should be generated from a single reproducible prediction file so that model tables and confusion matrices agree.

7.3 External validity

Performance on one language or dataset may not generalize to other languages, projects or development environments. Where possible, as part of evaluation, test across projects or, even better, across language(s). Use benchmark results from independent sources before claiming deployment.

7.4 Explanation validity and human evaluation

A clearly perceptible highlight is not equally honest to the model. Explanation faithfulness should be evaluated independently of readability. Human ratings may be influenced by programming experience, language familiarity, and expectations about AI. A real study should report participants' profiles, tasks, scales, and reviewer disagreements.

7.5 Synthetic-data constraint

The illustrative simulation results presented in this manuscript are not from any real-world model and do not support H1-H4. The model comparison and explanation ratings must be replaced with, or supplemented by, empirical results from a well-designed study before the manuscript can be considered a completed experimental validation.

8. Ethical and security issues

Using AI to assist with security vulnerability analysis may support defensive security, but source code and explanations may be sensitive in other contexts. Researchers should request permission before handling private repositories, adhere to all dataset licenses, and never send



private code to third-party systems without permission. Handle all results responsibly when they disclose previously unknown weaknesses; treat generated fixes as proposals until verified through testing, review, and security analysis.

Transparency also requires clear communication about the model's limits. Knowing which languages and code structures the model was trained on can help assess generalisability. It also helps to know whether a separate set of testing data was used in the evaluation. It can also help to know how the model's risk ratings were calibrated and how to interpret the explanation evidence.

9. Conclusion

This paper shows how to build an explanation-based AI for early software vulnerability detection. It covers source-code pre-processing, structural and semantic representation, classification, explanation generation, code localization, and developer review. The scientific literature on deep learning, graph-based analysis, code transformers, and explainability supports investigating this pairing, but also shows why predictor scores need careful handling.

Numerical tables show how to benchmark standard machine-learning models, CodeBERT, and an XAI-enhanced transformer, and how to report the usefulness of explanations. Since figures are simulated, treat them as a reporting example, not an empirical demonstration. The authors also need to address the discrepancy between the confusion-matrix metrics and the XAI-CodeBERT summary scores by inspecting the predictions before submission.

References

1. Arrieta, A. B., Díaz-Rodríguez, N., Del Ser, J., Bennetot, A., Tabik, S., Barbado, A., García, S., Gil-López, S., Molina, D., Benjamins, R., Chatila, R., & Herrera, F. (2020). Explainable artificial intelligence (XAI): Concepts, taxonomies, opportunities and challenges toward responsible AI. *Information Fusion*, 58, 82–115. <https://doi.org/10.1016/j.inffus.2019.12.012>
2. Chen, Z., Zhan, D., Yang, J., Yu, J., Chai, Y., Li, J., Li, G., & Wang, X. (2024). A survey of large language models for code intelligence. *ACM Computing Surveys*, 56(12), 1–38.
3. Chakraborty, S., Krishna, R., Ding, Y., & Ray, B. (2020). Deep learning-based vulnerability detection in source code: A survey. *ACM Computing Surveys*, 53(2), 1–36.
4. Doshi-Velez, F., & Kim, B. (2017). Towards a rigorous science of interpretable machine learning. *arXiv*. <https://arxiv.org/abs/1702.08608>
5. Fan, J., Li, Y., Wang, S., & Nguyen, T. N. (2020). A C/C++ code vulnerability dataset with code changes and CVE summaries. In *Proceedings of the 17th International Conference on Mining Software Repositories* (pp. 508–512). <https://doi.org/10.1145/3379597.3387501>
6. Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., Shou, L., Qin, B., Liu, T., Jiang, D., & Zhou, M. (2020). CodeBERT: A pre-trained model for programming and



- natural languages. In Findings of the Association for Computational Linguistics: EMNLP 2020 (pp. 1536–1547).
7. Guo, D., Ren, S., Lu, S., Duan, N., Svyatkovskiy, A., Fu, S., Tufano, M., Deng, S. K., Clement, C., Drain, D., Sundaresan, N., & Zhou, M. (2021). GraphCodeBERT: Pre-training code representations with data flow. In International Conference on Learning Representations.
 8. Guo, Y., Hu, Q., Tang, Q., & Le Traon, Y. (2023). An empirical study of the imbalance issue in software vulnerability detection. In Computer Security – ESORICS 2023 (pp. 371–390). Springer.
 9. Hassan, A. E. (2008). The road ahead for mining software repositories. In 2008 Frontiers of Software Maintenance (pp. 48–57).
 10. Jain, S., & Wallace, B. C. (2019). Attention is not explanation. In Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (pp. 3543–3556).
 11. Li, Z., Zou, D., Xu, S., Ou, X., Jin, H., & Zhu, Y. (2021). VulDeeLocator: A deep learning-based fine-grained vulnerability detection system. IEEE Transactions on Dependable and Secure Computing, 19(4), 2824–2838.
 12. Lundberg, S. M., & Lee, S.-I. (2017). A unified approach to interpreting model predictions. In Advances in Neural Information Processing Systems (Vol. 30).
 13. Mao, Q., Li, Z., Hu, X., Liu, K., Xia, X., & Sun, J. (2024). Towards explainable vulnerability detection with large language models. arXiv. <https://arxiv.org/abs/2406.09701>
 14. Patil, V.A., George, D., Dash, S. and Pasha, M.Y. (2026). AI for Digital Forensics and Cyber Threat Intelligence. In 2026 6th International Conference on Recent Trends in Computer Science and Technology (ICRTCST) (Vol. 1, pp. 459-465).
 15. Pearce, H., Ahmad, B., Tan, B., Dolan-Gavitt, B., & Karri, R. (2023). Asleep at the keyboard? Assessing the security of GitHub Copilot’s code contributions. Communications of the ACM, 67(1), 70–77.
 16. RB, S.R., Gomathi, T. and Shaby, S.M., 2025, May. Predicting software vulnerabilities with deep learning-driven static code analysis in security and cryptographic systems. In 2025 International Conference on Networks and Cryptology (NETCRYPT) (pp. 1677-1688).
 17. Ribeiro, M. T., Singh, S., & Guestrin, C. (2016). “Why should I trust you?” Explaining the predictions of any classifier. In Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (pp. 1135–1144).
 18. Russell, R., Kim, L., Hamilton, L., Lazovich, T., Harer, J., Ozdemir, O., Ellingwood, P., & McConley, M. (2018). Automated vulnerability detection in source code using deep representation learning. In Proceedings of the 17th IEEE International Conference on Machine Learning and Applications (pp. 757–762).



19. Sharma, K.P., Nagpal, T., Vora, T., Yadav, A., Abdullah, M.I., Jayaprakash, B., Kashyap, A., Sridevi, G., Bhowmik, A. and Bukate, B.B., 2025. Interpretable intrusion detection for IoT environments using a self-attention-based explainable AI framework. *Scientific Reports*, 15(1), p.39937.
20. Scandariato, R., Walden, J., Hovsepyan, A., & Joosen, W. (2014). Predicting vulnerable software components via text mining. *IEEE Transactions on Software Engineering*, 40(10), 993–1006.
21. Shin, Y., Meneely, A., Williams, L., & Osborne, J. A. (2011). Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities. *IEEE Transactions on Software Engineering*, 37(6), 772–787.
22. Sommestad, T., Ekstedt, M., & Johnson, P. (2013). A probabilistic relational model for security risk analysis of enterprise architecture. *Information and Software Technology*, 55(1), 16–28.
23. Sundararajan, M., Taly, A., & Yan, Q. (2017). Axiomatic attribution for deep networks. In *Proceedings of the 34th International Conference on Machine Learning* (pp. 3319–3328).
24. Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., & Polosukhin, I. (2017). Attention is all you need. In *Advances in Neural Information Processing Systems* (Vol. 30).
25. Walden, J., Doyle, J., Rice, M., & Bogdanoski, M. (2014). The impact of a major security flaw on software development practices. *IEEE Software*, 31(3), 43–49.
26. White, M., Tufano, M., Vendome, C., & Poshyvanyk, D. (2019). Deep learning code fragments for code clone detection. In *Proceedings of the 31st IEEE International Conference on Software Maintenance and Evolution* (pp. 87–97).
27. Xu, F. F., He, J., Neubig, G., & Srikumar, V. (2022). A systematic evaluation of large language models of code. In *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming* (pp. 1–10).
28. Zhang, J., Wang, X., Zhang, D., & Chen, H. (2023). Explainable artificial intelligence for cybersecurity: A systematic review. *IEEE Transactions on Artificial Intelligence*, 4(6), 1322–1338.
29. Zhou, Y., Liu, S., Siow, J., Du, X., & Liu, Y. (2019). Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. In *Advances in Neural Information Processing Systems* (Vol. 32, pp. 10197–10207).
30. Zou, D., Wang, S., Xu, S., Li, Z., & Jin, H. (2019). μ VulDeePecker: A deep learning-based system for multiclass vulnerability detection. *IEEE Transactions on Dependable and Secure Computing*, 18(5), 2224–2236.
31. Zimmermann, T., Nagappan, N., Gall, H., Giger, E., & Murphy, B. (2009). Cross-project defect prediction: A large scale experiment on data vs. domain vs. process. In *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium* (pp. 91–100).