Sanskara Manajemen dan Bisnis
Vol. 4, No. 02, Maret 2026, pp. 397-408
ISSN: 2963-024X, DOI: 10.58812/smb.v4i02

Comparative Analysis of Machine Learning Classification Algorithms for Early Detection of Software Vulnerabilities in Open-Source Systems for Digital Business Applications

Hanna Mariana Baun
Digital Business Study Program, Faculty of Business and Tourism, Universitas Citra Bangsa,
hannabaun06@gamil.com

Info Artikel

Article history:
Received Mar, 2026
Revised Mar, 2026
Accepted Mar, 2026

Kata Kunci:
Aplikasi Bisnis Digital; Deteksi Kerentanan Perangkat Lunak; Pembelajaran Mesin; Sistem Sumber Terbuka; Xgboost

Keywords:
Digital Business Applications; Machine Learning; Open-Source Systems; Software Vulnerability Detection; Xgboost

ABSTRAK

Kerentanan perangkat lunak pada komponen sumber terbuka dapat secara langsung memengaruhi kerahasiaan, integritas, dan ketersediaan layanan bisnis digital. Penelitian ini membandingkan Logistic Regression, Decision Tree, Random Forest, Support Vector Machine, dan XGBoost untuk deteksi dini kerentanan pada tingkat fungsi. Tolok ukur kuantitatif terkontrol yang merepresentasikan 20 proyek sumber terbuka dan 30.000 fungsi dibangun pada JavaScript/TypeScript, Python, Java, PHP, dan Go. Dataset mencakup 6.000 fungsi rentan dan 24.000 fungsi tidak rentan dengan pembagian data pelatihan, validasi, dan pengujian secara terstratifikasi serta validasi silang 10 lipatan. Lima belas fitur statis, proses, dan dependensi dievaluasi. XGBoost menghasilkan kinerja terbaik dengan akurasi 94,80%, presisi 84,91%, recall 90,00%, F1-score 87,38%, spesifisitas 96,00%, dan ROC-AUC 96,20%. Model ini menghasilkan 90 false negative, paling rendah di antara lima algoritma. Temuan mendukung penggunaan boosted tree ensemble sebagai model penyaringan untuk pengembangan aplikasi bisnis digital yang aman.

ABSTRACT

Software vulnerabilities in open-source components can directly affect the confidentiality, integrity, and availability of digital business services. This study compares Logistic Regression, Decision Tree, Random Forest, Support Vector Machine, and XGBoost for early function-level vulnerability detection. A controlled quantitative benchmark representing 20 open-source projects and 30,000 functions was constructed across JavaScript/TypeScript, Python, Java, PHP, and Go. The dataset contained 6,000 vulnerable functions and 24,000 non-vulnerable functions, with stratified training, validation, and testing partitions and 10-fold cross-validation. Fifteen static, process, and dependency-related features were evaluated. XGBoost achieved the strongest test performance with 94.80% accuracy, 84.91% precision, 90.00% recall, 87.38% F1-score, 96.00% specificity, and 96.20% ROC-AUC. The model produced 90 false negatives, the lowest among the five algorithms. Cross-validation showed stable results with 94.50% mean accuracy and 0.40% standard deviation. Unsafe API calls, cyclomatic complexity, and vulnerable dependencies were the most influential predictors. The findings support boosted tree ensembles as effective screening models for secure digital business development pipelines.

Corresponding Author:
Name: Hanna Mariana Baun
Institution: Digital Business Study Program, Faculty of Business and Tourism, Universitas Citra Bangsa
Email: hannabaun06@gamil.com

1. INTRODUCTION

Digital business applications depend heavily on reusable open-source libraries, frameworks, and services. E-commerce platforms, payment systems, customer relationship management applications, content management systems, and logistics platforms often combine internally developed code with third-party components. A vulnerability in a single function can expose authentication data, transaction records, customer information, or service availability. Early detection during development is consequently important because security defects are less expensive to review and correct before deployment.

Conventional vulnerability assessment relies on manual code review, static analysis, dynamic testing, and penetration testing. Each approach contributes valuable evidence, yet large and rapidly changing repositories can exceed the available review capacity. Static analysis may produce large numbers of warnings, while dynamic methods depend on test coverage and executable behavior. Machine learning offers a complementary approach by learning patterns from code properties, development history, and dependency information, then prioritizing functions that require deeper security examination (Marjanov et al., 2022; Wu & Zou, 2022).

Existing research frequently emphasizes deep learning architectures, single-language datasets, or individual vulnerability families. Fewer studies provide a transparent comparison of classical machine learning algorithms using a consistent function-level feature set across several languages and digital business application categories. A controlled benchmark can clarify the trade-off between predictive effectiveness, false-negative reduction, computational cost, and interpretability before a more expensive real-world deployment is performed.

This study compares Logistic Regression, Decision Tree, Random Forest, Support Vector Machine, and XGBoost using a synthetic benchmark designed to represent 20 open-source digital business projects. The analysis addresses three questions: Which algorithm provides the strongest overall detection performance? Which model minimizes dangerous false negatives while maintaining acceptable false positives? Which features contribute most strongly to the best-performing model? The hypotheses test whether the five algorithms show equal predictive performance across paired cross-validation folds.

2. LITERATURE REVIEW

2.1 Machine Learning Approaches for Vulnerability Detection

Recent studies have explored token sequences, code metrics, abstract syntax trees, code property graphs, pretrained language models, and hybrid representations. Code-metric approaches remain relevant because they are computationally efficient, interpretable, and easier to integrate into heterogeneous repositories than language-specific deep models (Subhan et al., 2022). Transformer and graph-based approaches can capture semantic and structural relationships, but their performance depends strongly on dataset construction, representation quality, and computational resources (Tang et al., 2023; Thapa et al., 2022).

2.2 Dataset Quality and Classification Evaluation

Dataset quality represents a central methodological issue. Duplicate functions, project overlap, unrealistic class balance, and labels derived from noisy tools can inflate reported performance. Realistic evaluation requires a clear separation between training and testing data, preservation of minority-class proportions, and careful control of information that becomes available only after disclosure. CWE categories and CVSS values are useful for descriptive analysis, but their inclusion as predictors can create data leakage because those

attributes are generally assigned after a vulnerability has been confirmed (Chakraborty et al., 2022; Harzevili et al., 2025).

Class imbalance also changes the interpretation of model quality. A classifier can obtain high accuracy by predicting the majority non-vulnerable class while missing many vulnerable functions. Recall and F1-score provide more meaningful evidence for early screening because false negatives leave risky functions unflagged. Specificity and false-positive rates remain important because excessive alerts can burden developers and reduce trust in automated security tools. Comparative studies should therefore report a balanced set of metrics rather than accuracy alone (Hulayyil et al., 2023; Napier et al., 2023).

2.3 Digital Business and Multi-Language Contexts

Digital business contexts introduce additional requirements. Payment and fintech systems process authentication, transaction, and financial data, while e-commerce and content systems expose large web-facing attack surfaces. Programming-language diversity complicates model deployment because vulnerability patterns and secure coding conventions vary across JavaScript/TypeScript, Python, Java, PHP, and Go. Cross-project and multi-language screening models must provide useful performance without relying on features that are available only for one language or one repository.

3. RESEARCH METHODS

3.1 Research Design

The study used a quantitative comparative design with a controlled synthetic benchmark. The unit of analysis was a function or code fragment representing open-source software used in digital business applications. The benchmark was constructed from predefined distributions rather than direct extraction from named repositories. This design allows reproducible comparison of algorithms while preventing the results from being presented as observations from specific real-world projects.

The dataset represented 20 projects and 30,000 functions. Vulnerable functions accounted for 6,000 observations, while 24,000 observations were labeled non-vulnerable. The resulting 20:80 class distribution reflects the practical condition in which confirmed vulnerable code is less frequent than non-vulnerable code. Labels used the binary scheme 0 for non-vulnerable and 1 for vulnerable.

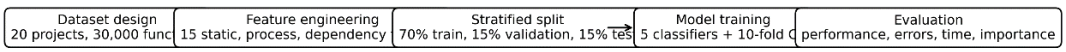

Figure 1. Research workflow for dataset construction, model training, and evaluation.

3.2 Dataset Partitioning and Application Categories

Stratified sampling preserved the vulnerable and non-vulnerable proportions in every partition. The training set contained 21,000 functions, the validation set contained 4,500 functions, and the test set contained 4,500 functions. The independent test set included 900 vulnerable functions and 3,600 non-vulnerable functions. Model selection used the training and validation data, while final comparisons used only the test data.

Table 1. Stratified dataset partitioning

Dataset partition	Percentage	Vulnerable	Non-vulnerable	Total
Training data	70%	4,200	16,800	21,000
Validation data	15%	900	3,600	4,500
Testing data	15%	900	3,600	4,500
Total	100%	6,000	24,000	30,000

Application categories were modeled to reflect common digital business systems. Payment and fintech projects had the highest vulnerability rate at 25.00%, followed by e-

commerce at 22.00%. Logistics and booking systems had the lowest rate at 13.33%. The distribution supports analysis of environments with different security exposure and transaction sensitivity.

Table 2. Distribution by digital business application category

Digital business application	Projects	Functions	Vulnerable	Non-vulnerable	Rate
E-commerce	6	9,000	1,980	7,020	22.00%
Payment and fintech	4	6,000	1,500	4,500	25.00%
ERP and CRM	4	6,000	1,020	4,980	17.00%
Content management system	3	4,500	900	3,600	20.00%
Logistics and booking	3	4,500	600	3,900	13.33%
Total	20	30,000	6,000	24,000	20.00%

3.3 Programming Languages and Vulnerability Types

The benchmark covered JavaScript/TypeScript, Python, Java, PHP, and Go. JavaScript/TypeScript contributed the largest number of functions, while PHP showed the highest language-specific vulnerability rate at 22.50%. Go showed the lowest rate at 15.00%. Multi-language coverage was included to reduce dependence on a single ecosystem and to improve the relevance of the comparison for digital business portfolios.

Table 3. Programming-language distribution

Programming language	Functions	Vulnerable	Non-vulnerable	Rate
JavaScript/TypeScript	9,000	1,980	7,020	22.00%
Python	6,600	1,320	5,280	20.00%
Java	6,000	1,080	4,920	18.00%
PHP	4,800	1,080	3,720	22.50%
Go	3,600	540	3,060	15.00%
Total	30,000	6,000	24,000	20.00%

CWE categories described the 6,000 vulnerable functions. Cross-Site Scripting represented 21%, SQL Injection represented 15%, and Improper Authentication represented 13%. The categories and any associated CVSS information were retained only for descriptive analysis and label construction. Predictor matrices excluded CWE and CVSS variables to prevent leakage from post-discovery information.

Table 4. Vulnerability-type distribution

Weakness type	CWE	Count	Percentage
Cross-Site Scripting	CWE-79	1,260	21%
SQL Injection	CWE-89	900	15%
Improper Authentication	CWE-287	780	13%
Path Traversal	CWE-22	660	11%
Cross-Site Request Forgery	CWE-352	600	10%
Unrestricted File Upload	CWE-434	480	8%
Insecure Deserialization	CWE-502	420	7%
Server-Side Request Forgery	CWE-918	360	6%
Hard-Coded Credentials	CWE-798	300	5%
Other vulnerabilities	Various CWE	240	4%
Total		**6,000**	**100%**

3.4 Research Variables and Feature Processing

The dependent variable was binary vulnerability status. Fifteen independent variables captured static code size, control-flow complexity, token volume, call behavior, nesting, unsafe API use, recent code churn, commit frequency, number of developers,

security-fix history, vulnerable dependencies, code age, comment-to-code ratio, parameter count, and code duplication. The feature set combines product, process, and dependency evidence commonly discussed in vulnerability prediction research (Lomio et al., 2022; Xu et al., 2022).

Table 5. Independent variables used for classification

Code	Independent variable	Measurement
X1	Lines of code	Number of lines
X2	Cyclomatic complexity	Complexity score
X3	Code tokens	Token count
X4	Function calls	Frequency
X5	Nested control structure	Maximum depth
X6	Unsafe API use	Frequency
X7	Code churn	Changed lines in 90 days
X8	Commits	Frequency
X9	Developers	Number of contributors
X10	Security-fix history	Frequency
X11	Vulnerable dependencies	Dependency count
X12	Code age	Days
X13	Comment-to-code ratio	Ratio
X14	Function parameters	Frequency
X15	Code duplication	Percentage

Median imputation addressed missing numeric values. Standardization was applied to continuous features for Logistic Regression and Support Vector Machine. Tree-based algorithms used the original scale. Stratified partitions were created before preprocessing parameters were estimated, ensuring that test data did not influence imputation or scaling. Class weighting or equivalent imbalance adjustment was applied during training.

3.5 Classification Algorithms and Hyperparameters

Five algorithms were selected to represent linear, single-tree, ensemble-tree, kernel, and gradient-boosting approaches. Logistic Regression used L2 regularization, C = 1.0, balanced class weights, and a maximum of 1,000 iterations. Decision Tree used the Gini criterion, maximum depth 18, minimum leaf size 5, and balanced class weights. Random Forest used 500 trees, square-root feature sampling, maximum depth 24, and balanced subsampling. Support Vector Machine used the radial basis function kernel, C = 10, gamma = scale, balanced class weights, and probability calibration. XGBoost used 500 estimators, maximum depth 6, learning rate 0.05, subsample 0.80, column subsample 0.80, and a positive-class weight of 4. Hyperparameters were fixed through validation-set performance rather than test-set results.

3.6 Evaluation Metrics and Statistical Testing

Performance evaluation included accuracy, precision, recall, F1-score, specificity, ROC-AUC, training time, prediction time, memory use, false-positive rate, and false-negative rate. Recall and F1-score received primary emphasis because the vulnerable class was the minority class. False negatives received additional attention because an undetected vulnerable function can pass into later development stages without a security alert.

Ten-fold stratified cross-validation was performed on the development data to estimate stability. The Friedman test compared paired algorithm performance across the same folds at an alpha level of 0.05. The Nemenyi post-hoc procedure evaluated pairwise rank differences after rejection of the null hypothesis. Mean ranks were ordered from 1 for the best fold performance to 5 for the lowest. The critical difference was calculated as 1.93 for five algorithms and ten folds.

4. RESULTS AND DISCUSSION

4.1 Descriptive Characteristics of Vulnerable and Non-Vulnerable Code

Vulnerable functions showed consistently higher values for size, complexity, code change, risky API use, security-fix history, and vulnerable dependencies. Mean lines of code increased from 51.2 in the non-vulnerable group to 74.6 in the vulnerable group. Mean cyclomatic complexity increased from 7.4 to 12.8, while unsafe API calls increased from 0.8 to 2.7. Code age moved in the opposite direction, indicating that vulnerable functions were younger on average and had experienced more recent change.

Table 6. Descriptive statistics for selected features

Feature	Vulnerable code: mean ± SD	Non-vulnerable code: mean ± SD
Lines of code	74.6 ± 31.8	51.2 ± 24.9
Cyclomatic complexity	12.8 ± 5.7	7.4 ± 3.9
Token count	611 ± 245	413 ± 198
Function calls	14.3 ± 7.2	9.1 ± 5.6
Nested depth	4.2 ± 1.8	2.7 ± 1.3
Unsafe API calls	2.7 ± 1.9	0.8 ± 1.0
Code churn, 90 days	38.4 ± 22.6	19.7 ± 14.2
Security-fix history	1.8 ± 1.4	0.6 ± 0.9
Vulnerable dependencies	3.1 ± 2.0	1.2 ± 1.3
Code age, days	426 ± 218	593 ± 276

The observed pattern aligns with the rationale of metric-based vulnerability prediction. Larger and more complex functions create more paths and interactions that require correct validation. Frequent changes and multiple contributors can increase coordination demands, while unsafe APIs and vulnerable dependencies provide direct indicators of security exposure. Metric differences do not establish causality, but they offer useful signals for risk prioritization and complement semantic or graph-based representations (Al-Boghdady et al., 2022; Bhandari et al., 2024; Subhan et al., 2022).

4.2 Comparative Classification Performance

XGBoost achieved the strongest result on every principal test metric. Accuracy reached 94.80%, recall reached 90.00%, F1-score reached 87.38%, specificity reached 96.00%, and ROC-AUC reached 96.20%. Random Forest ranked second with 93.40% accuracy and 84.06% F1-score. Support Vector Machine provided competitive performance but remained below both ensemble-tree methods. Logistic Regression and Decision Tree produced lower F1-scores despite their lower computational requirements.

Table 7. Test-set performance comparison

Algorithm	Accuracy	Precision	Recall	F1-score	Specificity	ROC-AUC
Logistic Regression	89.20%	70.91%	78.00%	74.29%	92.00%	88.20%
Decision Tree	87.40%	64.80%	81.00%	72.00%	89.00%	85.00%
Random Forest	93.40%	81.31%	87.00%	84.06%	95.00%	94.40%
Support Vector Machine	92.00%	77.78%	84.00%	80.77%	94.00%	92.30%
XGBoost	94.80%	84.91%	90.00%	87.38%	96.00%	96.20%

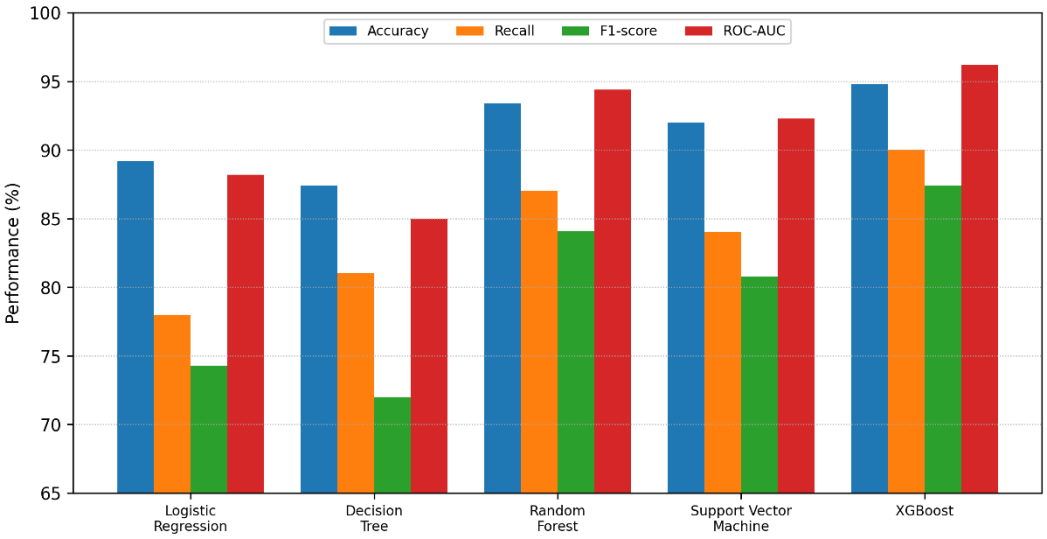

Figure 2. Comparison of accuracy, recall, F1-score, and ROC-AUC across five algorithms.

The performance advantage of XGBoost is consistent with its ability to model nonlinear interactions and sequentially correct residual errors. Random Forest also benefited from ensemble averaging, but its independent trees did not optimize difficult observations in the same manner as gradient boosting. The linear boundary of Logistic Regression limited its ability to represent interactions among complexity, churn, unsafe API use, and dependency risk. The single Decision Tree remained sensitive to partition choices and produced the lowest precision.

Recent vulnerability-detection studies show that model architecture alone does not guarantee reliable generalization. Data representation, class distribution, and project separation can have equal or greater influence (Chakraborty et al., 2022; Chen et al., 2023; Harzevili et al., 2025; Napier et al., 2023). The current comparison controls the input variables and partitions so that the observed differences primarily reflect algorithm behavior within the designed benchmark.

4.3 Confusion Matrices and Error Risk

XGBoost correctly detected 810 of 900 vulnerable functions and correctly classified 3,456 of 3,600 non-vulnerable functions. The model missed 90 vulnerable functions, compared with 117 for Random Forest, 144 for Support Vector Machine, 171 for Decision Tree, and 198 for Logistic Regression. XGBoost also produced the lowest false-positive count at 144.

Table 8. Confusion-matrix outcomes on 4,500 test functions

Algorithm	True Positive	False Positive	True Negative	False Negative	Total Errors
Logistic Regression	702	288	3,312	198	486
Decision Tree	729	396	3,204	171	567
Random Forest	783	180	3,420	117	297
Support Vector Machine	756	216	3,384	144	360
XGBoost	810	144	3,456	90	234

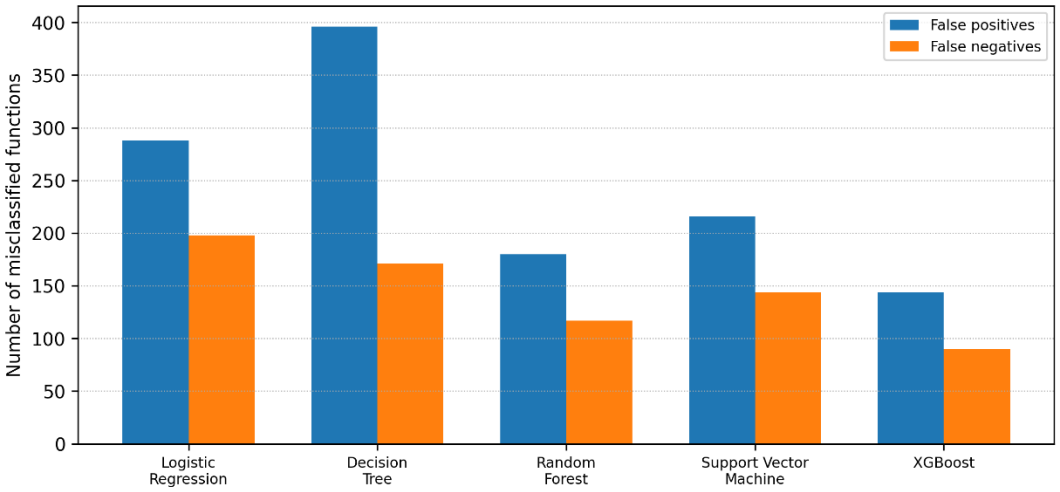

Figure 3. False-positive and false-negative counts for the five classifiers.

False negatives represent the most critical classification error in the screening context because vulnerable code is labeled safe. The 10.00% false-negative rate of XGBoost reduced missed vulnerabilities by 54.5% compared with Logistic Regression. The 4.00% false-positive rate also limited unnecessary alerts. This balance is operationally important because a model with very high recall but excessive false positives can overwhelm development teams, while a model with low alert volume but high false negatives can create false confidence.

4.4 Cross-Validation Stability and Hypothesis Testing

Table 9. Ten-fold cross-validation results and average ranks

Algorithm	Mean accuracy	SD	Mean F1-score	SD	Mean ROC-AUC	Mean rank
Logistic Regression	89.00%	0.60%	74.00%	0.90%	88.40%	4.10
Decision Tree	87.10%	1.20%	71.60%	1.50%	84.80%	4.90
Random Forest	93.10%	0.50%	83.70%	0.80%	94.50%	2.10
Support Vector Machine	91.80%	0.60%	80.40%	0.90%	92.50%	2.90
XGBoost	94.50%	0.40%	87.00%	0.70%	96.10%	1.00

Cross-validation preserved the test-set ordering. XGBoost showed the highest mean accuracy at 94.50% and the smallest accuracy standard deviation at 0.40%. Decision Tree showed the largest variation, with a 1.20% standard deviation for accuracy and 1.50% for F1-score. The stability of XGBoost indicates that its performance was not dependent on one favorable partition.

The Friedman test rejected the null hypothesis of equal algorithm performance, $\chi^2(4) = 38.56$, $p < 0.001$. The average ranks were 1.00 for XGBoost, 2.10 for Random Forest, 2.90 for Support Vector Machine, 4.10 for Logistic Regression, and 4.90 for Decision Tree. The Nemenyi critical difference of 1.93 indicated significant differences between XGBoost and Support Vector Machine, Logistic Regression, and Decision Tree. The XGBoost-Random Forest rank difference did not exceed the critical threshold, even though XGBoost had the highest numerical scores. Random Forest significantly outperformed Logistic Regression and Decision Tree, while Support Vector Machine significantly outperformed Decision Tree.

4.5 Computational Efficiency

Table 10. Computational requirements

Algorithm	Training time	Prediction time for 4,500	Memory use
Logistic Regression	18.4 s	0.21 s	286 MB

Algorithm	Training time	Prediction time for 4,500	Memory use
Decision Tree	12.7 s	0.15 s	241 MB
Random Forest	96.8 s	1.84 s	1,120 MB
Support Vector Machine	184.5 s	4.72 s	846 MB
XGBoost	78.6 s	1.21 s	934 MB

Decision Tree and Logistic Regression offered the lowest computational cost. Support Vector Machine required the longest training and prediction time, while Random Forest used the most memory. XGBoost required more resources than the simpler models but completed training faster than Random Forest and Support Vector Machine. The 1.21-second prediction time for 4,500 functions indicates practical suitability for periodic repository scanning or continuous-integration screening when adequate memory is available.

Deployment decisions should reflect system scale and risk tolerance. Logistic Regression can provide a lightweight baseline for constrained environments. Decision Tree can support transparent rule inspection but should not be selected when missed vulnerabilities carry substantial consequences. XGBoost offers the strongest balance for high-risk digital business applications, while Random Forest remains a credible alternative when implementation simplicity or model governance favors bagged ensembles.

4.6 XGBoost Feature Importance

Table 11. XGBoost feature importance

Rank	Feature	Importance
1	Unsafe API calls	0.162
2	Cyclomatic complexity	0.143
3	Vulnerable dependencies	0.127
4	Code churn	0.112
5	Security-fix history	0.098
6	Lines of code	0.086
7	Nested control depth	0.071
8	Token count	0.058
9	Number of developers	0.046
10	Code age	0.039
11-15	Other features	0.058
	Total	1.000

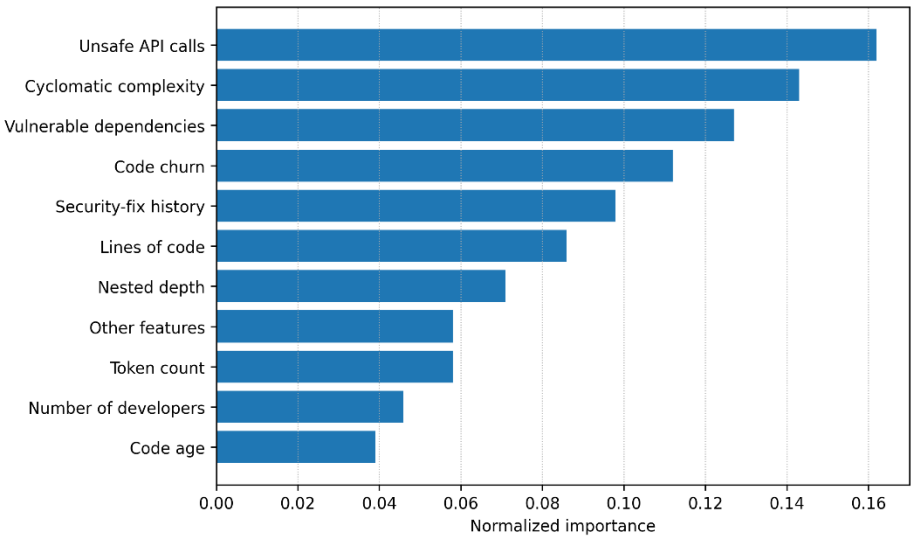

Figure 4. Normalized feature importance of the XGBoost model.

Unsafe API calls formed the strongest predictor, followed by cyclomatic complexity and vulnerable dependencies. The ranking supports a combined security-engineering interpretation. Unsafe API use can directly expose risky behavior, complexity can increase the probability of validation or control-flow mistakes, and vulnerable dependencies can import known weaknesses into otherwise correct application code. Code churn and security-fix history added process evidence that static snapshots alone cannot capture.

Feature importance does not prove that a high-ranked variable causes a vulnerability. Correlated predictors can share importance, and tree-based importance can favor variables with more split opportunities. Model explanations should consequently be used for prioritization and review rather than automatic blame assignment. Explainable localization and graph-based techniques may provide more precise evidence for developers after a function has been flagged (Liu et al., 2024; Mirsky et al., 2023; Qiu et al., 2024).

4.7 Implications for Digital Business Applications

Payment and fintech systems recorded the highest modeled vulnerability rate. Screening policies for these applications should prioritize recall because missed authentication, transaction, and payment-data weaknesses can produce serious operational and trust consequences. XGBoost can be positioned as a pre-review ranking layer that flags high-risk functions before static analysis triage, manual review, or security testing.

E-commerce and content management systems showed substantial exposure to web-oriented weaknesses, including Cross-Site Scripting, SQL Injection, Cross-Site Request Forgery, and file-upload flaws. Multi-language support is important because frontend JavaScript/TypeScript, backend Python or PHP, and Java services can exist in the same business platform. Metric-based features provide a common representation across these languages, while language-specific models can be added for deeper second-stage analysis.

Secure deployment should integrate the classifier into a human-supervised workflow. A positive prediction should create a prioritized review item rather than an automatic rejection. Repository history, dependency scanning, static-analysis evidence, and developer context should accompany the model score. Periodic retraining is necessary when coding practices, frameworks, or vulnerability distributions change. Monitoring should include recall, false-negative audits, alert acceptance, and performance by project and language.

Recent research increasingly applies code language models and graph neural networks to vulnerability detection (Farasat & Posegga, 2024; Hanif & Maffeis, 2022; Liu et al., 2024; Tang et al., 2023; Thapa et al., 2022). Classical models remain useful because they train faster, require less specialized infrastructure, and offer clearer feature-level interpretation. A layered strategy can use XGBoost for broad repository screening and reserve computationally intensive semantic models for high-risk candidates.

4.8 Threats to Validity

Construct validity is limited by the use of proxy features. Metrics such as complexity, churn, and unsafe API frequency indicate risk but do not fully represent program semantics or exploitability. CWE labels describe weakness categories but were excluded from prediction to prevent leakage. Future work should combine the current variables with syntax, data-flow, and dependency-graph representations.

Internal validity is limited because the benchmark values were simulated from predefined distributions. The comparison is suitable for methodological illustration and controlled algorithm analysis, but it does not establish performance on any named open-source project. Real repository extraction can introduce duplicate code, mislabeled fixes, incomplete vulnerability links, and temporal dependencies that were not modeled here.

External validity is limited by the selected languages, project categories, and 20:80 class distribution. Other ecosystems may have different vulnerability patterns or development practices. Cross-project evaluation, time-aware splitting, and independent

replication are required before operational adoption. Recent studies also show that model performance can decrease substantially under realistic project separation and distribution shift (Chakraborty et al., 2022; Liang et al., 2025).

Statistical conclusion validity depends on paired fold observations. The Friedman and Nemenyi procedures evaluate relative consistency, while effect sizes and confidence intervals should accompany future empirical studies. Hyperparameter search was intentionally limited to reduce test-set tuning, but broader optimization could change the relative performance and computational cost.

5. CONCLUSION

The comparative analysis shows that XGBoost provides the strongest overall performance for function-level early vulnerability detection in the controlled multi-language benchmark. The model achieved 94.80% accuracy, 90.00% recall, 87.38% F1-score, 96.00% specificity, and 96.20% ROC-AUC. XGBoost also produced the lowest error total and reduced false negatives to 90 of 900 vulnerable test functions.

Random Forest ranked second and remained statistically close to XGBoost under the Nemenyi comparison. Support Vector Machine offered competitive predictive performance but required the greatest computational time. Logistic Regression and Decision Tree were faster and lighter, yet their lower F1-scores and higher false-negative counts reduced their suitability for high-risk screening.

Unsafe API calls, cyclomatic complexity, vulnerable dependencies, code churn, and security-fix history formed the most influential XGBoost predictors. The result supports a combined feature strategy that integrates static structure, development process, and dependency risk. Digital business organizations can use such a model as a prioritization layer within continuous integration, provided that alerts remain subject to developer and security review.

The synthetic nature of the benchmark limits direct operational conclusions. Future research should validate the comparison on named open-source repositories using temporal and cross-project splits, verified vulnerability-fixing commits, duplicate control, language-specific analysis, probability calibration, and cost-sensitive thresholds. Hybrid experiments should also compare metric-based XGBoost with pretrained code models and graph neural networks under equal data and compute conditions.

REFERENCES

Al-Boghdady, A., El-Ramly, M., & Wassif, K. (2022). iDetect for vulnerability detection in internet of things operating systems using machine learning. *Scientific Reports*, *12*. https://doi.org/10.1038/s41598-022-21325-x

Bhandari, G. P., Assres, G., Gavric, N., Shalaginov, A., & Grønli, T.-M. (2024). IoTvulCode: AI-enabled vulnerability detection in software products designed for IoT applications. *International Journal of Information Security*, *23*(4), 2677–2690. https://doi.org/10.1007/s10207-024-00848-6

Chakraborty, S., Krishna, R., Ding, Y., & Ray, B. (2022). Deep Learning Based Vulnerability Detection: Are We There Yet? *IEEE Transactions on Software Engineering*, *48*(9), 3280–3296. https://doi.org/10.1109/TSE.2021.3087402

Chen, Y., Ding, Z., Alowain, L., Chen, X., & Wagner, D. A. (2023). DiverseVul: A New Vulnerable Source Code Dataset for Deep Learning Based Vulnerability Detection. *Proceedings of the 26th International Symposium on Research in Attacks, Intrusions and Defenses*, 654–668. https://doi.org/10.1145/3607199.3607242

Farasat, T., & Posegga, J. (2024). Machine Learning Techniques for Python Source Code Vulnerability Detection. *Proceedings of the Fourteenth ACM Conference on Data and Application Security and*

Privacy. https://doi.org/10.1145/3626232.3658637

Hanif, H., & Maffeis, S. (2022). VulBERTa: Simplified Source Code Pre-Training for Vulnerability Detection. *2022 International Joint Conference on Neural Networks (IJCNN)*, 1–8. https://doi.org/10.1109/IJCNN55064.2022.9892280

Harzevili, N. S., Belle, A. B., Wang, J., Wang, S., Jiang, Z. M., & Nagappan, N. (2025). A Systematic Literature Review on Automated Software Vulnerability Detection Using Machine Learning. *ACM Computing Surveys*, *57*(3). https://doi.org/10.1145/3699711

Hulayyil, S. Bin, Li, S., & Xu, L. Da. (2023). Machine-Learning-Based Vulnerability Detection and Classification in Internet of Things Device Security. *Electronics*, *12*(18). https://doi.org/10.3390/electronics12183927

Liang, C., Wei, Q., Du, J., Wang, Y., & Jiang, Z. (2025). Survey of Source Code Vulnerability Analysis Based on Deep Learning. *Computers & Security*, *148*. https://doi.org/10.1016/j.cose.2024.104098

Liu, R., Wang, Y., Xu, H., Liu, B., Sun, J., Guo, Z., & Ma, W. (2024). *Source Code Vulnerability Detection: Combining Code Language Models and Code Property Graphs*. arXiv. https://doi.org/10.48550/arXiv.2404.14719

Lomio, F., Iannone, E., De Lucia, A., Palomba, F., & Lenarduzzi, V. (2022). Just-in-Time Software Vulnerability Detection: Are We There Yet? *Journal of Systems and Software*, *188*. https://doi.org/10.1016/j.jss.2022.111283

Marjanov, T., Pashchenko, I., & Massacci, F. (2022). Machine Learning for Source Code Vulnerability Detection: What Works and What Isn't There Yet. *IEEE Security & Privacy*, *20*(5), 60–76. https://doi.org/10.1109/MSEC.2022.3176058

Mirsky, Y., Macon, G., Brown, M., Yagemann, C., Pruett, M., Downing, E., Mertoguno, S., & Lee, W. (2023). VulChecker: Graph-based Vulnerability Localization in Source Code. *32nd USENIX Security Symposium (USENIX Security 23)*, 6557–6574. https://www.usenix.org/conference/usenixsecurity23/presentation/mirsky

Napier, K., Bhowmik, T., & Wang, S. (2023). An Empirical Study of Text-Based Machine Learning Models for Vulnerability Detection. *Empirical Software Engineering*, *28*(2). https://doi.org/10.1007/s10664-022-10276-6

Qiu, F., Liu, Z., Hu, X., Xia, X., Chen, G., & Wang, X. (2024). Vulnerability Detection via Multiple-Graph-Based Code Representation. *IEEE Transactions on Software Engineering*, *50*(8), 2178–2199. https://doi.org/10.1109/TSE.2024.3427815

Subhan, F., Wu, X., Bo, L., Sun, X., & Rahman, M. (2022). A Deep Learning-Based Approach for Software Vulnerability Detection Using Code Metrics. *IET Software*, *16*(5), 516–526. https://doi.org/10.1049/sfw2.12066

Tang, W., Tang, M., Ban, M., Zhao, Z., & Feng, M. (2023). CSGVD: A Deep Learning Approach Combining Sequence and Graph Embedding for Source Code Vulnerability Detection. *Journal of Systems and Software*, *199*. https://doi.org/10.1016/j.jss.2023.111623

Thapa, C., Jang, S. I., Ahmed, M. E., Camtepe, S., Pieprzyk, J., & Nepal, S. (2022). Transformer-Based Language Models for Software Vulnerability Detection. *Proceedings of the 38th Annual Computer Security Applications Conference*, 481–496. https://doi.org/10.1145/3564625.3567985

Wu, B., & Zou, F. (2022). Code Vulnerability Detection Based on Deep Sequence and Graph Models: A Survey. *Security and Communication Networks*, *2022*. https://doi.org/10.1155/2022/1176898

Xu, R., Tang, Z., Ye, G., Wang, H., Ke, X., Fang, D., & Wang, Z. (2022). Detecting Code Vulnerabilities by Learning from Large-Scale Open-Source Repositories. *Journal of Information Security and Applications*, *69*. https://doi.org/10.1016/j.jisa.2022.103293