International Journal of Creative and Open Research in Engineering and Management
ISSN: **3108-1754 (Online)**
Volume 02 Issue 08 Augustl-2026 | Impact Factor: 3.5

Static Code Analysis Framework for Automated Security Vulnerability Detection

Nagaraju Vassey[1], Vani Pasupula[2], Manne Naga VJ Manikanth[3]

[1] Assistant Professor ,Department of Information technology and Computer Applications, Andhra University College of Engineering, Visakhapatnam, India
[2] MCA student ,Department of Information technology and Computer Applications, Andhra University College of Engineering, Visakhapatnam, India
[3] Guest faculty , Department of Information technology and Computer Applications, Andhra University College of Engineering, Visakhapatnam, India
Corresponding Author Email: manikanthmanne@gmail.com | ORCID: https://orcid.org/0009-0009-6557-4041

How to Cite this Article:

Pasupula, V. & Manikanth, M. N. V. (2026). Static Code Analysis Framework for Automated Security Vulnerability Detection. International Journal of Creative and Open Research in Engineering and Management, 02(8), 1-9. https://doi.org/10.55041/ijcope.v2i8.112

https://doi.org/10.55041/ijcope.v2i8.112

Abstract— The proliferation of software security vulnerabilities in modern applications has created an urgent demand for automated, intelligent, and scalable detection systems. Existing rule-based static analysis tools are effective for predefined vulnerability patterns but may have difficulty with variations that are not adequately covered by their rules, and a substantial portion of existing machine-learning-based vulnerability detection research focuses on C and C++ programs, while Python-specific approaches remain comparatively less explored. This paper presents a static code analysis framework for automated security vulnerability detection specifically targeting Python source code. The proposed framework implements a three-phase experimental pipeline: Phase 1 establishes a token-based TF-IDF baseline by representing source code as typed token pairs and training classical machine learning classifiers; Phase 2 introduces up to 35 hand-engineered structural features extracted from the Abstract Syntax Tree (AST) of each code snippet using the NodeVisitor design pattern; and Phase 3 combines both feature types into a unified representation and retrains all classifiers — Random Forest, Gradient Boosting, Logistic Regression, and Support Vector Machine. The system is trained and evaluated on a manually constructed, balanced dataset of 70 Python code snippets covering five high-severity CWE vulnerability categories: SQL Injection (CWE-89), OS Command Injection (CWE-78), Path Traversal (CWE-22), Insecure Deserialization (CWE-502), and Cross-Site Scripting (CWE-79). Recall is designated as the primary optimization metric because missed vulnerabilities carry greater risk than false alarms in security-critical deployment contexts. Experimental results show that AST-based structural features substantially improve recall compared with the TF-IDF baseline, while the combined TF-IDF and AST representation maintains this improved performance. The complete system is deployed as a command-line prediction tool producing a binary verdict, a continuous risk score, and a structured report of detected dangerous API patterns, and runs entirely on standard consumer hardware without GPU or deep-learning infrastructure requirements.

Keywords— Static Code Analysis; Vulnerability Detection; Python Security; Abstract Syntax Tree; TF-IDF; Machine Learning; CWE; Random Forest; Support Vector Machine.

1.INTRODUCTION

Software applications are now central to sectors such as banking, healthcare, education, e-commerce, and government services, and a single software vulnerability can compromise confidential data, interrupt essential services, and cause significant financial loss [14]. Software vulnerabilities are commonly identified using static analysis, which examines source code without executing the program, and dynamic analysis, which evaluates a program at runtime. Static analysis can be integrated into continuous development workflows and detect issues earlier, but existing tools such as Bandit, SonarQube, and Semgrep rely primarily on manually defined rules. Rule-based approaches are effective for well-established coding mistakes but struggle with novel attack techniques, variations of known vulnerabilities, or deliberately obfuscated code, and maintaining large rule sets becomes increasingly difficult as languages and threats evolve.

Machine learning offers a complementary approach by learning vulnerability patterns directly from labelled examples instead of relying exclusively on manually defined rules, potentially enabling classification of previously unseen source-code samples. The effectiveness of such models depends heavily on how source code is represented: token-based features (e.g., TF-IDF) capture the lexical vocabulary of a program but not its logical structure, whereas Abstract Syntax Tree (AST) features

preserve syntactic relationships, function definitions, control-flow statements, and API usage. Combining lexical and structural features can therefore produce a richer representation and improve a model's ability to distinguish vulnerable from secure code.

Python is one of the most widely adopted programming languages, used extensively in web development, automation, data science, and cloud computing, yet it remains susceptible to serious vulnerabilities such as SQL Injection (CWE-89), OS Command Injection (CWE-78), Path Traversal (CWE-22), Insecure Deserialization (CWE-502), and Cross-Site Scripting (CWE-79) [8]. Although machine learning has been widely investigated for vulnerability detection, a substantial portion of existing research focuses on C and C++, where memory-related vulnerabilities dominate, leaving Python comparatively less explored — particularly with respect to combining lexical and structural features. This paper addresses that gap by proposing a lightweight static analysis framework that integrates TF-IDF token features with AST-based structural features and employs classical machine learning algorithms to classify Python source code as Vulnerable or Safe, prioritizing Recall as the primary evaluation metric since a missed vulnerability is more costly than a false alarm in security-critical contexts, consistent with widely referenced industry vulnerability rankings such as the OWASP Top 10 [13].

The specific objectives of this work are to: (i) construct a labelled dataset of secure and vulnerable Python snippets across five CWE categories; (ii) build a TF-IDF token-feature baseline; (iii) design an AST-based structural feature extractor; (iv) train and evaluate Random Forest, Gradient Boosting, Logistic Regression, and Support Vector Machine classifiers on structural features; (v) combine token and structural features into a unified representation; (vi) compare classifiers using Accuracy, Precision, Recall, F1-Score, and AUC-ROC, with Recall given primary importance; and (vii) deliver a prediction tool that classifies unseen Python code, produces a risk score, and reports the detected security-relevant patterns.

2. LITERATURE REVIEW

Scandariato et al. [1] presented one of the earliest applications of machine learning to vulnerability prediction, treating source files as text documents and using a bag-of-words representation with Random Forest and Naïve Bayes classifiers on Java-based Android projects. The study established that lexical representations contain useful security signal, but the bag-of-words approach ignores syntax, control structures, and program behaviour, and the findings do not directly generalize beyond Java.

Pattan et al. [2] investigated Abstract Syntax Trees as a structural representation for Python source code, extracting features describing functions, loops, conditionals, operators, and assignments, and training an optimized Random Forest classifier. The work demonstrated that AST-based features provide a richer representation than purely token-based approaches, but it targeted general code classification rather than security-specific vulnerability detection.

Zhou et al. [3] proposed Devign, a graph neural network framework that integrates AST, Control Flow Graph, and Data Flow Graph representations to detect vulnerabilities in C programs, achieving strong performance but requiring large labelled datasets, GPU hardware, and producing representations that are difficult to interpret. Li et al. [4] proposed VulDeePecker, which extracts “code gadgets” around security-sensitive API calls and classifies them using a Bidirectional LSTM network; the approach achieved high accuracy on C/C++ code but similarly depends on large training sets and substantial computational resources, and is not directly applicable to Python.

Table 1. Comparison of Existing Studies

Study	Language	Feature Representation	Learning Technique	Limitation
Scandariato et al. [1] (2014)	Java	Token-based features	Random Forest, Naive Bayes	Ignores structural information
Pattan et al. [2] (2025)	Python	AST features	Random Forest	General code classification only
Zhou et al. [3] (2019) — Devign	C	AST + CFG + DFG	Graph Neural Network	High computational cost
Li et al. [4] (2018) — VulDeePecker	C/C++	Code Gadgets + Word Embeddings	BLSTM	Requires large datasets

Overall, the literature shows a shift from lexical to structural and, more recently, graph/deep-learning representations, with detection performance generally improving as more structural and semantic information is captured. However, most deep-learning approaches require large labelled datasets and heavy computational infrastructure [15] and are typically evaluated on C, C++, or Java rather than Python. Few studies combine lexical and structural features within a lightweight, interpretable, classical machine-learning framework specifically for Python — a gap this work addresses.

3. METHODOLOGY

The proposed framework is an automated, machine-learning-based static analysis system that accepts Python source code and determines whether it is Vulnerable or Safe without executing it. The workflow is organized into six stages: (1) dataset construction, (2) data preprocessing, (3) TF-IDF feature extraction, (4) AST feature extraction, (5) model training and evaluation, and (6) vulnerability prediction. To isolate the contribution of each feature representation, the study follows a three-phase experimental methodology: Phase 1 evaluates lexical information via TF-IDF token features using Logistic Regression, Random Forest, and Gradient Boosting; Phase 2 evaluates structural information extracted from the AST using

all four classifiers introduced in Section 5; and Phase 3 combines both representations into a unified feature vector and again evaluates all four classifiers. The Support Vector Machine classifier is not included among the reported Phase 1 results (see Section 6.1).

In addition to a binary Vulnerable/Safe verdict, the system produces a continuous risk score between 0 and 1 and reports the detected security-relevant patterns — such as dangerous API calls, untrusted input sources, and insecure function usage — identified during AST analysis, giving developers interpretable context for each prediction. The classifier achieving the highest Recall across the experimental phases, while maintaining balanced overall performance, is selected as the final deployment model and is exposed through a command-line prediction tool that supports demo, file, and inline code analysis modes.

4. DATASET AND FEATURE EXTRACTION

4.1 Dataset Construction

Since no publicly available dataset matched the selected Python vulnerability categories, a custom dataset of 70 Python code snippets was manually constructed — 35 vulnerable and 35 safe. The dataset was constructed to provide both vulnerable and secure Python examples across the selected CWE categories while maintaining balanced class representation. The dataset was designed as a controlled proof-of-concept dataset to evaluate the contribution of lexical and structural features under balanced experimental conditions, and does not claim to represent the full diversity of real-world Python vulnerabilities. The dataset spans five CWE categories [8], [9], summarized in Table 2.

Table 2. Vulnerability Categories Covered by the Dataset

CWE	Vulnerability	Secure Alternative
CWE-89	SQL Injection	Parameterized queries
CWE-78	OS Command Injection	subprocess.run()
CWE-22	Path Traversal	Path validation
CWE-502	Insecure Deserialization	json.loads()
CWE-79	Cross-Site Scripting	Escaped template rendering

Each snippet was manually labelled (1 = Vulnerable, 0 = Safe) and stored in a CSV file. Preprocessing removed Python comments, validated source-code syntax, and applied a stratified 80:20 train-test split (56 training / 14 testing samples) using a fixed random seed of 42 to preserve class balance and ensure reproducibility.

4.2 TF-IDF Token Features (Phase 1)

Python's tokenize module [6] converts each source file into a sequence of programming tokens (keywords, identifiers, operators, literals). These tokens are transformed into numerical vectors using Term Frequency-Inverse Document Frequency (TF-IDF) with a maximum vocabulary of 500 features via TfidfVectorizer, emphasizing informative and security-relevant tokens while down-weighting common ones.

4.3 AST-Based Structural Features (Phase 2)

Each file is parsed with Python's ast.parse() [5] into an Abstract Syntax Tree, which is traversed by a custom VulnerabilityFeatureExtractor (derived from ast.NodeVisitor) that computes structural features across eight categories, summarized in Table 3. The feature extractor is designed to support a maximum of 35 structural features; however, for the dataset constructed in this study, only 34 of these features were active — that is, present with non-zero variation — in the resulting feature representation, and therefore contributed to the final feature matrix. The AST representation used for model training was consequently 34-dimensional. This distinction reflects the composition of the constructed dataset rather than an error in the feature-extraction implementation. Extracted features are normalized with StandardScaler prior to training.

Table 3. AST Structural Feature Categories

Category	Examples
Dangerous APIs	os.system(), eval(), pickle.loads()
Taint Sources	input(), request.args
Structural Complexity	Functions, classes, imports
Control Flow	if, loops, try-except
Validation Signals	Comparisons, assertions
Data Structures	Lists, dictionaries
Return/Yield	return, yield
Scope & Logic	lambda, global, comprehensions

4.4 Combined Feature Representation (Phase 3)

Phase 3 merges the two representations into a single feature vector for each program. The TF-IDF vectorizer produced 452 features from the training data (below the 500-feature cap, given the small dataset), which were concatenated with the 34 active AST features described in Section 4.3 to form a 486-dimensional hybrid representation, using SciPy sparse-matrix operations and preserving both lexical and structural information for each sample.

5.MACHINE LEARNING MODELS

Four supervised classifiers, representing diverse learning paradigms in classical machine learning [15], were evaluated using standard Scikit-learn [7] implementations with default parameters (extensive hyperparameter tuning was avoided given the small dataset, to reduce overfitting risk): Logistic Regression as a linear baseline, Random Forest [11] as an ensemble of decision trees, Gradient Boosting [12] as a sequential ensemble of weak learners, and Support Vector Machine (RBF) [10] for non-linear decision-boundary learning. These classifiers and their purposes are summarized in Table 4.

Table 4. Machine Learning Classifiers Evaluated

Classifier	Purpose
Logistic Regression	Linear baseline classifier
Random Forest	Ensemble learning using multiple decision trees
Gradient Boosting	Sequential boosting of weak learners
Support Vector Machine (RBF)	Non-linear decision-boundary learning

Model training used 5-fold Stratified Cross-Validation to preserve class distribution across folds, followed by evaluation on an independent hold-out test set. Performance was assessed using Accuracy, Precision, Recall, F1-Score, and AUC-ROC, with Recall treated as the primary criterion because, in security-critical settings, a false negative (a missed vulnerability) is considerably more costly than a false positive.

6. RESULTS AND DISCUSSION

6.1 Phase 1 — TF-IDF Baseline

Table 5 reports the Phase 1 results for Logistic Regression, Random Forest, and Gradient Boosting. The Support Vector Machine classifier, although evaluated in Phases 2 and 3, is not included in the Phase 1 results because no corresponding Phase 1 SVM result exists in the underlying experimental records; no such result is reported here in order to avoid implying a finding that was not obtained.

Table 5. Performance Using TF-IDF Token Features (Phase 1)

Classifier	Accuracy	Precision	Recall	F1-Score	AUC-ROC
Logistic Regression	0.5714	0.6667	0.2857	0.4000	0.6531
Random Forest	0.6429	0.6667	0.5714	0.6154	0.8367
Gradient Boosting	0.7857	1.0000	0.5714	0.7273	0.8673

Gradient Boosting achieved the best Phase 1 performance (Accuracy 78.57%, Precision 100%, F1 0.7273, AUC-ROC 0.8673), but its Recall of 57.14% shows that three of seven vulnerable test samples were missed. Logistic Regression performed weakest (Recall 28.57%), indicating that a linear decision boundary is insufficient for the lexical feature space. These results confirm that token-level information alone is a useful but limited baseline, motivating the structural feature extraction explored next.

6.2 Phase 2 — AST Structural Features

Table 6. Performance Using AST Structural Features (Phase 2)

Classifier	Accuracy	Precision	Recall	F1-Score	AUC-ROC
Random Forest	0.9286	1.0000	0.8571	0.9231	1.0000
Gradient Boosting	1.0000	1.0000	1.0000	1.0000	1.0000
Logistic Regression	1.0000	1.0000	1.0000	1.0000	1.0000
SVM (RBF)	0.9286	1.0000	0.8571	0.9231	0.9796

AST-based features substantially improved performance over the TF-IDF baseline. Gradient Boosting and Logistic Regression achieved perfect scores on every metric, and Random Forest and SVM each missed only a single vulnerable sample. These results confirm that structural information — how programming constructs are used, not merely whether they occur — provides markedly more discriminative signal for vulnerability detection than lexical frequency alone.

6.3 Phase 3 — Combined TF-IDF + AST Features

Table 7. Performance Using Combined TF-IDF + AST Features (Phase 3)

Classifier	Accuracy	Precision	Recall	F1-Score	CV F1	AUC-ROC
Logistic Regression	1.0000	1.0000	1.0000	1.0000	0.9106	1.0000
Random Forest	0.7857	0.8333	0.7143	0.7692	0.8044	0.9592
Gradient Boosting	1.0000	1.0000	1.0000	1.0000	0.8749	1.0000
SVM (RBF)	0.9286	1.0000	0.8571	0.9231	0.9260	0.9796

Logistic Regression and Gradient Boosting again achieved perfect test-set metrics; because Logistic Regression obtained a higher cross-validation F1-score (0.9106 vs. 0.8749), it was selected as the final deployment model. Random Forest performed comparatively weaker on the larger 486-dimensional hybrid space (Recall 71.43%), suggesting it benefited less from the additional lexical dimensions than the linear and boosting models. Across all three phases, Recall improved from a best of 57.14% (Phase 1) to a perfect 100% (Phases 2 and 3 for the best classifiers). This improvement is primarily attributable to the AST-based structural features introduced in Phase 2; the combined representation evaluated in Phase 3 maintains this level of performance while keeping the pipeline computationally lightweight and free of GPU or deep-learning infrastructure requirements.

6.4 Overall Comparison and Cautious Interpretation

Table 8 summarizes the best reported performance obtained for each feature representation across the classifiers evaluated in the corresponding phase.

Table 8. Best Reported Performance by Feature Representation

Representation	Best Accuracy	Best Precision	Best Recall	Best F1
TF-IDF (Phase 1)	0.7857	1.0000	0.5714	0.7273
AST (Phase 2)	1.0000	1.0000	1.0000	1.0000
TF-IDF + AST (Phase 3)	1.0000	1.0000	1.0000	1.0000

TF-IDF provides the baseline lexical representation, correctly capturing some but not all vulnerability signal. AST provides markedly stronger structural information in this experiment, raising every metric to a perfect or near-perfect score for the best classifiers. The combined representation maintains the improved performance obtained with AST but does not demonstrate an additional performance improvement over AST alone in this dataset — Table 8 shows identical best-case values for Phases 2 and 3. The perfect test-set scores obtained by some classifiers should be interpreted cautiously because the evaluation dataset contains only 14 held-out samples. Although stratified cross-validation was used during model development, evaluation on larger and externally sourced datasets is necessary to establish generalization beyond the controlled dataset used here.

7.LIMITATIONS

• The experimental dataset contains only 70 manually labelled Python snippets, which limits training diversity and may affect generalization to large, real-world codebases.

• The framework performs binary classification only (Vulnerable/Safe) and does not identify the specific vulnerability category or the exact vulnerable line of code.

• Detection is restricted to five CWE categories (SQL Injection, OS Command Injection, Path Traversal, Insecure Deserialization, and Cross-Site Scripting); other vulnerability types, such as buffer overflows, insecure authentication, or SSRF, are not covered.

• The framework supports Python only; extending it to other languages such as Java, C++, or JavaScript would require redesigning feature extraction and retraining the models.

• Being a static analysis approach, the system cannot detect vulnerabilities that depend on runtime execution, program state, user interaction, or environmental conditions.

• AST feature extraction requires syntactically valid Python code; files with syntax errors or incomplete fragments cannot be parsed and analyzed.

• Each snippet is analyzed independently, without interprocedural, data-flow, or cross-module dependency analysis, which may be needed for more complex, multi-file vulnerabilities.

• Results are based on a fixed train-test split and random seed; performance may vary with different data partitions, and evaluation on larger external datasets would strengthen the evidence for generalizability.

8. CONCLUSION

This paper presented a three-phase static code analysis framework for automated security vulnerability detection in Python source code, combining TF-IDF lexical features with AST-based structural features within a classical machine-learning pipeline. The experimental evaluation shows that token-based features provide a useful but limited baseline, that AST-based structural features substantially improve detection performance in this controlled experiment, and that the combined feature representation maintains this improved performance without demonstrating a further gain over AST alone. Among all evaluated classifiers, Logistic Regression trained on the combined feature representation achieved perfect Accuracy, Precision, Recall, F1-Score, and AUC-ROC on the test set while also achieving the highest cross-validation F1-score (0.9106) among the combined-representation models, and was selected as the deployment model behind a command-line

prediction tool that reports a Vulnerable/Safe verdict, a risk score, and the detected security-relevant patterns. These results should be interpreted cautiously given the small, manually constructed dataset and the 14-sample held-out test set; larger, real-world datasets are required before stronger conclusions about generalization can be drawn. Future work includes enlarging and diversifying the dataset with real-world repositories, extending coverage to additional CWE categories and programming languages, supporting multi-class and line-level vulnerability localization, and comparing the proposed approach against transformer-based models such as CodeBERT.

REFERENCES

[1] R. Scandariato, J. Walden, A. Hovsepyan, and W. Joosen, "Predicting Vulnerable Software Components via Text Mining," IEEE Transactions on Software Engineering, vol. 40, no. 10, pp. 993-1006, Oct. 2014.

[2] S. F. K. Pattan, S. Sudharson, and S. Mynampati, "Optimizing Code Feature Extraction and Classification Using Random Forest Models on Abstract Syntax Trees," in Proc. IEEE 9th Int. Conf. on Information and Communication Technology (CICT), 2025, pp. 1-7.

[3] Y. Zhou, S. Liu, J. Siow, X. Du, and Y. Liu, "Devign: Effective Vulnerability Identification by Learning Comprehensive Program Semantics via Graph Neural Networks," arXiv preprint arXiv:1909.03496, 2019.

[4] Z. Li, D. Zou, S. Xu, X. Ou, H. Jin, S. Wang, Z. Deng, and Y. Zhong, "VulDeePecker: A Deep Learning-Based System for Vulnerability Detection," in Proc. Network and Distributed System Security (NDSS) Symposium, 2018.

[5] Python Software Foundation, "ast - Abstract Syntax Trees," Python 3 Documentation. [Online]. Available: https://docs.python.org/3/library/ast.html

[6] Python Software Foundation, "tokenize - Tokenizer for Python Source," Python 3 Documentation. [Online]. Available: https://docs.python.org/3/library/tokenize.html

[7] F. Pedregosa et al., "Scikit-learn: Machine Learning in Python," Journal of Machine Learning Research, vol. 12, pp. 2825-2830, 2011.

[8] MITRE Corporation, "Common Weakness Enumeration (CWE)." [Online]. Available: https://cwe.mitre.org

[9] National Institute of Standards and Technology (NIST), "National Vulnerability Database (NVD)." [Online]. Available: https://nvd.nist.gov

[10] C. Cortes and V. Vapnik, "Support-Vector Networks," Machine Learning, vol. 20, no. 3, pp. 273-297, 1995.

[11] L. Breiman, "Random Forests," Machine Learning, vol. 45, no. 1, pp. 5-32, 2001.

[12] J. H. Friedman, "Greedy Function Approximation: A Gradient Boosting Machine," The Annals of Statistics, vol. 29, no. 5, pp. 1189-1232, 2001.

[13] OWASP Foundation, "OWASP Top 10: The Ten Most Critical Web Application Security Risks." [Online]. Available: https://owasp.org/www-project-top-ten/

[14] G. McGraw, Software Security: Building Security In. Boston, MA, USA: Addison-Wesley Professional, 2006.

[15] T. Hastie, R. Tibshirani, and J. Friedman, The Elements of Statistical Learning, 2nd ed. New York, NY, USA: Springer, 2009.