World Journal of Advanced Research and Reviews

eISSN: 2581-9615 CODEN (USA): WJARAI

Cross Ref DOI: 10.30574/wjarr

Journal homepage: https://wjarr.com/

(Research Article)

Source code vulnerability identification using machine learning models

Utoda Reuben [1], Tom Innocent [1,*] and Jerry Ogar [2]

[1] *Department of Computer Science, University of Cross River State Calabar, Nigeria.*

[2] *Department of Electrical Electronics, University of Cross River State Calabar, Nigeria.*

World Journal of Advanced Research and Reviews, 2026, 30(03), 1372-1391

Publication history: Received on 10 May 2026; revised on 14 June 2026; accepted on 17 June 2026

Article DOI: https://doi.org/10.30574/wjarr.2026.30.3.1713

Abstract

Software vulnerabilities remain a major challenge in modern software development, frequently leading to security breaches, unauthorized access, service disruption, and financial losses. Although traditional vulnerability detection methods such as static and dynamic analysis are widely used, they often generate excessive false positives and struggle to capture complex patterns within source code. Recent advances in machine learning provide an opportunity to improve vulnerability detection by automatically learning relationships that are difficult to identify using rule-based approaches.

This study presents a machine learning framework for automated source code vulnerability identification using benchmark datasets obtained from the Software Assurance Reference Dataset (SARD) and the National Vulnerability Database (NVD). The framework includes source code preprocessing, feature extraction, model training, hyperparameter optimization, and performance evaluation. Four models Decision Tree, Random Forest, Support Vector Machine, and Long Short-Term Memory (LSTM) were implemented and evaluated using accuracy, precision, recall, and F1-score.

Experimental results show that LSTM achieved the best overall performance with an accuracy of 93.4%, followed by Random Forest at 91.2%. These findings indicate that models capable of learning contextual and sequential information are particularly effective for vulnerability detection. The proposed framework demonstrates how machine learning can support secure software development by reducing manual analysis effort and enabling scalable vulnerability assessment. Its potential application extends to continuous security monitoring and integration within modern software development pipelines.

Keywords: Source Code Security; Vulnerability Detection; Machine Learning; Deep Learning; Software Engineering; Secure Coding

1. Introduction

Software systems underpin critical services across healthcare, finance, transportation, cloud computing, manufacturing, and industrial control environments. As these systems continue to increase in size, complexity, and interconnectivity, software security has become a major concern for organizations and developers. Vulnerabilities embedded within source code remain one of the most frequently exploited attack vectors, often leading to data breaches, unauthorized access, service disruptions, financial losses, and reputational damage. The growing reliance on open-source components, third-party libraries, and interconnected software ecosystems has further complicated the task of identifying and mitigating security weaknesses during software development [1], [2].

* Corresponding author: Tom Innocent

Copyright © 2026 Author(s) retain the copyright of this article. This article is published under the terms of the Creative Commons Attribution License 4.0.

Traditional vulnerability detection techniques, including static code analysis, dynamic testing, and manual code review, play an important role in software assurance. However, these approaches often generate large numbers of false positives, require significant expert intervention, and struggle to identify vulnerabilities that depend on complex semantic and contextual relationships within source code. As modern software systems become increasingly heterogeneous and distributed, the limitations of purely rule-based detection mechanisms have become more evident [3].

Recent advances in machine learning (ML) and deep learning (DL) have created new opportunities for automated vulnerability detection. Unlike conventional approaches that rely on predefined rules and signatures, learning-based models can automatically discover patterns from large collections of vulnerable and non-vulnerable source code samples. Studies have demonstrated the effectiveness of Convolutional Neural Networks (CNNs), Long Short-Term Memory (LSTM) networks, Graph Neural Networks (GNNs), and transformer-based architectures for identifying software vulnerabilities with improved accuracy and scalability [4], [6]. By learning syntactic, structural, and semantic characteristics directly from source code, these approaches can detect vulnerability patterns that may be overlooked by traditional analysis techniques.

Despite significant progress, several challenges remain. Vulnerability datasets are frequently imbalanced, model performance often varies across programming languages and vulnerability categories, and many learning-based approaches provide limited interpretability of prediction outcomes [2], [4]. Furthermore, the effectiveness of vulnerability detection systems is strongly influenced by feature representation strategies, dataset quality, and experimental design [5]. These challenges continue to limit the practical deployment of learning-based vulnerability detection systems in real-world software engineering environments.

To address these challenges, this study develops and evaluates a machine learning framework for automated source code vulnerability identification. The framework compares traditional machine learning algorithms and sequence-based deep learning models under a unified experimental setting using benchmark datasets obtained from the Software Assurance Reference Dataset (SARD) and the National Vulnerability Database (NVD). By evaluating multiple learning paradigms using standardized preprocessing, feature extraction, hyperparameter optimization, and performance evaluation procedures, the study provides a comprehensive assessment of their effectiveness for vulnerability detection.

The proposed framework integrates software metrics, TF-IDF feature representations, and token-sequence-based deep learning within a reproducible evaluation environment. The framework facilitates systematic comparison of traditional machine learning and deep learning approaches while providing insights into the influence of feature representation on vulnerability detection performance and deployment suitability.

1.1. Research Gap and Contributions

Recent research has demonstrated the effectiveness of machine learning, deep learning, graph-based learning, and transformer-based architectures for source code vulnerability detection. However, direct comparison among these approaches remains difficult because many studies employ different datasets, preprocessing procedures, feature extraction techniques, evaluation metrics, and experimental configurations. In addition, many investigations focus primarily on advanced architectures while providing limited comparison with classical machine learning baselines under identical experimental conditions.

Consequently, there remains a need for a reproducible vulnerability detection framework that evaluates multiple learning paradigms using standardized datasets, preprocessing pipelines, optimization procedures, and performance metrics. Such a framework can provide a more objective assessment of the relative strengths and limitations of different approaches and facilitate informed deployment decisions in software engineering environments.

The major contributions of this study are:

- Development of a unified machine learning framework for source code vulnerability identification using benchmark datasets.
- Comparative evaluation of classical machine learning and deep learning models under identical experimental conditions.
- Investigation of the impact of feature representation on vulnerability detection performance.
- Assessment of model suitability for integration into secure software development and DevSecOps workflows.

2. Related Work

The increasing complexity of software systems and the growing frequency of cyberattacks have intensified research efforts toward automated source code vulnerability detection. Traditional vulnerability analysis techniques, including static analysis, dynamic analysis, and manual code inspection, have long been employed to identify software security flaws. While these methods remain important components of software assurance, they often generate high false positive rates, require substantial expert involvement, and exhibit limited capability in capturing complex semantic relationships within source code [7]. As a result, researchers have increasingly explored machine learning and deep learning approaches to improve vulnerability detection accuracy, scalability, and automation.

Early machine learning approaches relied heavily on handcrafted feature engineering. Software metrics such as cyclomatic complexity, control-flow characteristics, function-call dependencies, and code complexity measures were extracted and used as inputs to classification algorithms [8]. Models including Decision Trees (DT), Random Forests (RF), and Support Vector Machines (SVMs) demonstrated promising results in identifying vulnerable code components. However, these approaches depend strongly on feature quality and often fail to capture deeper syntactic and semantic properties embedded within source code [9].

The emergence of deep learning significantly advanced the field by enabling automatic feature learning from source code. Convolutional Neural Networks (CNNs) and Recurrent Neural Networks (RNNs), particularly Long Short-Term Memory (LSTM) architectures, have been successfully applied to vulnerability detection tasks by modeling source code as sequential data [10]. These models can capture contextual information and long-range dependencies within source code, thereby improving vulnerability identification performance compared with traditional machine learning approaches.

More recently, graph-based learning methods have attracted substantial attention. Researchers have employed Abstract Syntax Trees (ASTs), Control Flow Graphs (CFGs), Program Dependency Graphs (PDGs), and Code Property Graphs (CPGs) to preserve structural and semantic relationships within software systems [11]. Graph Neural Networks (GNNs) constructed on these representations have demonstrated strong capability in identifying complex vulnerabilities by learning interactions among program components. Comparative studies further indicate that incorporating structural information substantially enhances vulnerability detection accuracy and robustness [12].

The introduction of transformer-based architectures and Large Language Models (LLMs) has established a new direction for source code vulnerability analysis. Pre-trained models such as CodeBERT and related transformer frameworks have demonstrated remarkable effectiveness in learning programming language semantics and contextual dependencies [13]. Recent studies combining transformer models with graph-based representations have achieved state-of-the-art performance by simultaneously leveraging semantic and structural information [14]. Furthermore, emerging evidence suggests that LLMs possess considerable potential for both vulnerability detection and automated vulnerability remediation, although challenges related to computational cost, explainability, and cross-project generalization remain significant [9], [15].

Despite substantial progress, several research challenges persist. Existing approaches frequently encounter class imbalance issues, limited cross-language generalization, dataset quality concerns, and insufficient interpretability of prediction outcomes [4]. The availability of large-scale, accurately labeled, and diverse vulnerability datasets also remains a critical challenge affecting model robustness and practical applicability [2]. Additionally, increasing evidence suggests that hybrid approaches integrating software metrics, contextual code representations, and deep learning techniques may provide improved detection performance while reducing false positives [7], [14].

Motivated by these challenges, this study proposes a comparative machine learning framework for source code vulnerability identification using multiple learning algorithms. The framework seeks to evaluate the effectiveness of classical machine learning and sequence-based deep learning approaches under a unified experimental setting and identify the most suitable models for practical deployment in secure software engineering environments.

3. Methodology

This study proposes a machine learning-based framework for automated source code vulnerability identification. The methodology consists of six major stages: dataset collection, data preprocessing, feature extraction, model development, hyperparameter optimization, and performance evaluation. The framework is designed to accurately identify

vulnerable source code segments while ensuring model robustness, reproducibility, and generalizability. Figure 1 illustrates the overall workflow of the proposed methodology.

To prevent train-test contamination, duplicate source code fragments were removed before dataset partitioning. Additionally, all preprocessing and feature extraction operations were performed exclusively on training data before being applied to testing samples.

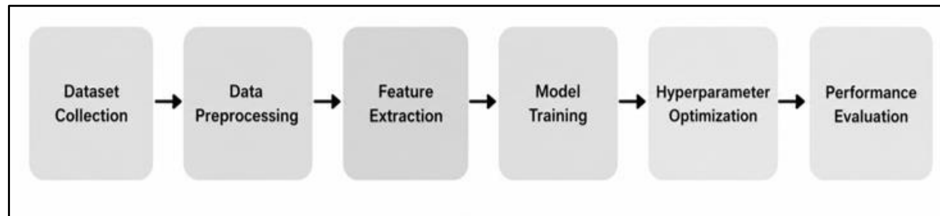

Figure 1 Framework workflow

3.1. Dataset Collection and Statistics

The experimental dataset was constructed using two publicly available benchmark repositories: the Software Assurance Reference Dataset (SARD) and the National Vulnerability Database (NVD).

3.1.1. Software Assurance Reference Dataset (SARD)

SARD, maintained by the National Institute of Standards and Technology (NIST), contains approximately 200,000 test programs representing more than 150 classes of software weaknesses. The repository includes source code written in multiple programming languages, including C, C++, Java, PHP, and C#. Common vulnerabilities represented in the dataset include:

- Buffer overflows
- SQL injection
- Cross-site scripting (XSS)
- Memory leaks
- Integer overflows
- Improper input validation

The availability of both vulnerable and secure implementations makes SARD suitable for supervised machine learning classification tasks.

3.1.2. National Vulnerability Database (NVD)

The National Vulnerability Database (NVD) contains publicly disclosed vulnerabilities reported under the Common Vulnerabilities and Exposures (CVE) program. Each vulnerability record includes:

- Vulnerability descriptions;
- Common Weakness Enumeration (CWE) classifications;
- Common Vulnerability Scoring System (CVSS) scores;
- Affected software products and versions;
- Security advisories and patch information.
- NVD provides real-world vulnerability information that complements the synthetic examples contained in SARD.

3.1.3. Dataset Construction Procedure

Source code samples were collected from the SARD repository and supplemented with vulnerability metadata obtained from NVD. Duplicate records were identified through hash-based matching and subsequently removed. Samples containing incomplete source code or ambiguous vulnerability labels were excluded. After preprocessing and filtering, a balanced subset of vulnerable and non-vulnerable source code samples was selected for experimentation.

3.1.4. Dataset Class Distribution

The original SARD and NVD repositories contain substantially larger numbers of records than those used in this study. To ensure computational feasibility, dataset balance, and consistent experimental evaluation, a subset of source code samples was selected. Following duplicate removal, preprocessing, and quality filtering, 40,000 samples were retained, consisting of 20,000 vulnerable and 20,000 non-vulnerable instances.

Table 1 Dataset Class Distribution

Class	Number of Samples	Percentage (%)
Vulnerable	20,000	50.0
Non-Vulnerable	20,000	50.0
Total	40,000	100.0

The balanced class distribution minimizes classification bias and enables fair model evaluation.

Table 2 Programming Language Distribution

Language	Samples
C	16,500
C++	10,200
Java	7,800
PHP	3,000
C#	2,500
Total	40,000

Table 3 Vulnerability Category Distribution

CWE Category	Samples
CWE-119 Buffer Overflow	8,500
CWE-89 SQL Injection	7,000
CWE-79 XSS	6,500
CWE-190 Integer Overflow	5,000
CWE-20 Input Validation	13,000

Because some source code samples contain multiple vulnerability categories, the cumulative frequency of vulnerability labels exceeds the total number of vulnerable instances.

3.1.5. Dataset Preparation

The original SARD and NVD repositories contain significantly larger numbers of records than those used in this study. To ensure computational efficiency and balanced classification, a subset of source code samples was selected. After duplicate removal, data cleaning, and filtering of incomplete records, 40,000 samples were retained for experimentation, comprising 20,000 vulnerable and 20,000 non-vulnerable instances.

3.2. Data Preprocessing

Raw source code often contains inconsistencies and irrelevant information that may negatively affect machine learning performance. Therefore, preprocessing was performed to transform source code into machine-readable representations.

3.2.1. Source Code Cleaning

The following cleaning operations were performed:

- Removal of comments and documentation strings;
- Elimination of unnecessary whitespace and blank lines;
- Removal of redundant metadata and compiler directives;
- Elimination of duplicate source code samples.

3.2.2. Lexical Tokenization

The cleaned source code was decomposed into lexical tokens consisting of:

- Keywords;
- Operators;
- Identifiers;
- Function names;
- Literals and constants;
- Delimiters.

The tokenized source code is represented as:

$$S = \{t_1, t_2, \dots, t_n\}$$

where:
S denotes the source code sample;
t_i denotes the ith token;
n represents the total number of tokens.

3.2.3. Code Normalization

To reduce vocabulary variability:

- Variable names were replaced with generic identifiers;
- Function names were standardized;
- Numeric values and string constants were normalized.

For example,

- **Original code**

```
int passwordLength = strlen(password);
```

- **Normalized code**

```
TYPE VAR1 = FUNC1(VAR2);
```

3.2.4. Sequence Padding

Because source code lengths vary significantly, token sequences were padded to a fixed length of **500 tokens**:

$$T_p = \begin{cases} \{t_1, \dots, t_m, 0, \dots, 0\}, & m < 500 \\ \{t_1, \dots, t_{500}\}, & m \geq 500 \end{cases}$$

where T_p denotes the padded sequence.

3.2.5. Label Encoding

Binary labels were assigned according to vulnerability status:

$$y = \begin{cases} 1, & \text{Vulnerable} \\ 0, & \text{Non-Vulnerable} \end{cases}$$

3.2.6. Data Standardization

Numerical features were standardized using:

$$z = \frac{x - \mu}{\sigma}$$

where:
x is the original feature value;
μ is the feature mean;
σ is the standard deviation.

The preprocessing workflow is summarized as:

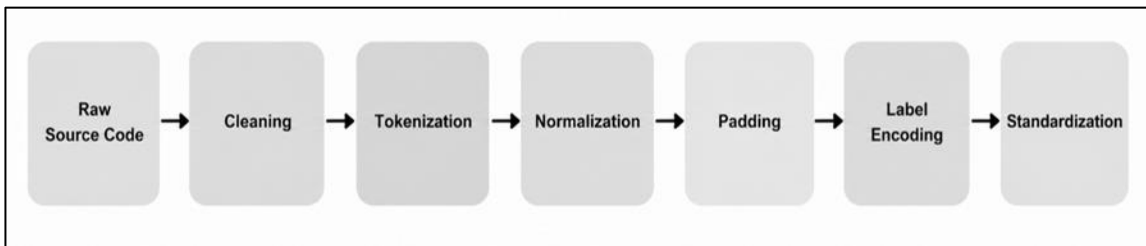

Figure 2 The preprocessing workflow

3.2.7. Prevention of Data Leakage

To ensure unbiased evaluation, duplicate source code fragments were removed before dataset partitioning. Training and testing datasets were separated prior to model development, and feature extraction parameters were derived exclusively from the training set before being applied to the testing data.

3.3. Feature Extraction

Feature extraction converts source code into numerical representations suitable for machine learning algorithms.

3.3.1. Software Metric Features

The following software metrics were extracted:

- Cyclomatic complexity (CC);
- Number of function calls (NFC);
- Number of loops (NL);
- Number of conditional statements (NCS);
- Number of input/output operations (NIO);
- Number of memory allocation operations (NMA);
- Source code length (LOC).

Cyclomatic complexity was computed as:

$$CC = E - N + 2P$$

where:

$E$is the number of edges;

$N$is the number of nodes;

$P$is the number of connected components.

Source code length was computed as:

$$LOC = \sum_{i=1}^{n} S_i$$

where $S_i$denotes the i^{th}executable statement.

3.3.2. Lexical Features

Lexical feature frequencies were computed as:

$$f(t_i) = \sum_{j=1}^{n} I(t_j = t_i)$$

where:

$$I(t_j = t_i) = \begin{cases} 1, & t_j = t_i \\ 0, & \text{otherwise} \end{cases}$$

3.3.3. TF-IDF Representation

Tokenized source code was transformed using TF-IDF:

$$TFIDF(t,d) = TF(t,d) \times \log\left(\frac{N}{DF(t)}\right)$$

where:

$TF(t,d)$denotes term frequency;

$DF(t)$denotes document frequency;

Nrepresents the total number of documents.

3.3.4. Feature Matrix Construction

The hybrid feature vector is represented as:

$$X = [x_1, x_2, \ldots, x_m]$$

The complete feature matrix is defined as:

$$F = \begin{bmatrix} x_{11} & x_{12} & \cdots & x_{1m} \\ x_{21} & x_{22} & \cdots & x_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ x_{n1} & x_{n2} & \cdots & x_{nm} \end{bmatrix}$$

where:

$n = 40{,}000$source code samples;

$m$denotes the total number of extracted features.

3.3.5. ture Utilization by Models

The classical machine learning models (Decision Tree, Random Forest, and Support Vector Machine) were trained using software metrics and TF-IDF feature representations. In contrast, the LSTM model utilized tokenized source code sequences as input through an embedding layer, enabling automatic learning of contextual and sequential patterns from source code.

Table 4 Feature Utilization by Models

Feature Type	DT	RF	SVM	LSTM
Software Metrics	✓	✓	✓	✗
TF-IDF Features	✓	✓	✓	✗
Token Sequences	✗	✗	✗	✓

3.3.6. Justification of Feature Representation

Although graph-based representations such as Abstract Syntax Trees (ASTs), Program Dependency Graphs (PDGs), and Code Property Graphs (CPGs) have demonstrated strong performance in vulnerability detection, the present study focuses on software metrics, TF-IDF features, and token sequences to establish a computationally efficient baseline suitable for practical deployment environments.

3.4. Machine Learning Models

Four classification models were implemented:

- Decision Tree (DT)
- Random Forest (RF)
- Support Vector Machine (SVM)
- Long Short-Term Memory (LSTM)

The DT, RF, and SVM models were implemented using Scikit-learn, while the LSTM model was implemented using TensorFlow-Keras.

3.5. Long Short-Term Memory Architecture

The LSTM model was designed to capture sequential dependencies and contextual information embedded in source code.

Token sequences were transformed into integer representations using a vocabulary generated from the training dataset. Unknown tokens encountered during testing were mapped to a dedicated out-of-vocabulary token. Sequences shorter than 500 tokens were padded with zeros, while longer sequences were truncated.

3.5.1. Input Representation

- Vocabulary size: 20,000 tokens;
- Maximum sequence length: 500 tokens;
- Embedding dimension: 128.

3.5.2. Proposed Architecture

Table 5 LSTM Architecture

Layer	Configuration
Input Layer	Sequence Length = 500
Embedding Layer	Vocabulary = 20,000; Dimension = 128

LSTM Layer 1	128 Units
Dropout Layer	0.30
LSTM Layer 2	64 Units
Dense Layer	64 Neurons (ReLU)
Dropout Layer	0.50
Output Layer	1 Neuron (Sigmoid)

3.5.3. Hyperparameters

Table 6 LSTM Hyperparameters

Parameter	**Value**
Optimizer	Adam
Learning Rate	0.001
Batch Size	64
Epochs	50
Loss Function	Binary Cross-Entropy
Validation Split	20%
Early Stopping Patience	5
Random Seed	42

The binary cross-entropy loss function is:

$$L = -\frac{1}{N}\sum_{i=1}^{N}[\, y_i \log(\hat{y}_i) + (1 - y_i)\log(1 - \hat{y}_i)]$$

3.6. Hyperparameter Optimization Using Grid Search

Grid Search with Stratified 10-Fold Cross-Validation was employed to optimize the machine learning models.

Table 7 Decision Tree Settings

Parameter	**Search Values**
Criterion	{gini, entropy}
Max Depth	{10, 20, 30, None}
Min Samples Split	{2, 5, 10}
Min Samples Leaf	{1, 2, 4}
Splitter	{best, random}

3.6.1. Optimal Parameters:

- Criterion = entropy
- Max Depth = 20
- Min Samples Split = 5
- Min Samples Leaf = 2

Splitter = best

Table 8 Random Forest Settings

Parameter	Search Values
n_estimators	{100, 200, 300, 500}
Criterion	{gini, entropy}
Max Depth	{10, 20, 30, None}
Min Samples Split	{2, 5, 10}
Min Samples Leaf	{1, 2, 4}
Max Features	{sqrt, log2}
Bootstrap	{True, False}

3.6.2. Optimal Parameters:

- n_estimators = 300
- Criterion = gini
- Max Depth = 30
- Max Features = sqrt
- Bootstrap = True

Table 9 Support Vector Machine Settings

Parameter	Search Values
Kernel	{linear, rbf, poly}
C	{0.1, 1, 10, 100}
Gamma	{scale, auto, 0.001, 0.01, 0.1}
Degree	{2, 3, 4}

3.6.3. Optimal Parameters:

Kernel = rbf

C = 10

Gamma = 0.01

3.6.4. Train/Test Split Information

Dataset Partitioning

The dataset was divided into training and testing sets using an 80:20 ratio. The training set contained 32,000 samples, while the testing set contained 8,000 samples. Stratified sampling was employed to preserve the class distribution across both subsets.

Table 10 Dataset Split

Dataset	Samples
Training	32,000
Testing	8,000
Total	40,000

3.7. Cross-Validation Statistics

A Stratified 10-Fold Cross-Validation procedure was adopted.

Table 11 Cross-Validation Distribution

Metric	Value
Total Samples	40,000
Number of Folds	10
Samples per Fold	4,000
Training Samples per Fold	36,000
Validation Samples per Fold	4,000
Vulnerable Samples per Fold	2,000
Non-Vulnerable Samples per Fold	2,000

The mean performance metric is calculated as:

$$\bar{x} = \frac{1}{k}\sum_{i=1}^{k} x_i$$

The standard deviation is calculated as:

$$\sigma = \sqrt{\frac{1}{k-1}\sum_{i=1}^{k}(x_i - \bar{x})^2}$$

where $k = 10$denotes the number of folds.

3.7.1. Hardware and Software Environment

Experiments were conducted using Python 3.11, Scikit-Learn 1.x, TensorFlow 2.x, and Keras. Model training was performed on a workstation equipped with Intel Core i7-12700K, 32 GB RAM, NVIDIA RTX 4070 (12 GB), Windows 11.

3.8. Performance Evaluation Metrics

The models were evaluated using Accuracy, Precision, Recall, and F1-Score.

Accuracy

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN}$$

Precision

$$Precision = \frac{TP}{TP + FP}$$

Recall

$$Recall = \frac{TP}{TP + FN}$$

F1-Score

$$F1 = 2\left(\frac{Precision \times Recall}{Precision + Recall}\right)$$

where:
TP = True Positives;
TN = True Negatives;
FP = False Positives;
FN = False Negatives.

Algorithm 1: Proposed Vulnerability Identification Framework

Input: Source code dataset D

Output: Vulnerability prediction labels

- Collect SARD and NVD datasets
- Preprocess source code samples
- Extract software metrics and TF-IDF features
- Split dataset into training and testing sets
- Optimize model parameters using Grid Search
- Train DT, RF, SVM, and LSTM models
- Perform 10-fold cross-validation
- Evaluate models using Accuracy, Precision,
- Recall, and F1-Score
- Select the best-performing model
- Generate vulnerability predictions

4. Results and Discussion

This section presents the experimental results obtained from the implementation of the proposed machine learning-based framework for source code vulnerability identification. The performance of four machine learning models Decision Tree (DT), Random Forest (RF), Support Vector Machine (SVM), and Long Short-Term Memory (LSTM) was evaluated using the test dataset.

4.1. Experimental Results

The evaluation was based on four performance metrics: accuracy, precision, recall, and F1-score. Table 12 summarizes the performance of the selected machine learning models.

Table 12 Performance Comparison of Machine Learning Models

Model	Accuracy (%)	Precision (%)	Recall (%)	F1-Score (%)
Decision Tree	82.4	80.1	78.9	79.5
Random Forest	91.2	89.8	90.5	90.1
Support Vector Machine	87.6	85.9	86.8	86.3
LSTM	93.4	92.1	91.7	91.9

The results indicate that the LSTM model achieved the highest overall performance with an accuracy of 93.4%, followed by Random Forest with 91.2%. The Decision Tree model achieved the lowest performance among the evaluated models, likely due to its tendency to overfit training data and its limited capability to capture complex semantic relationships within source code.

Table 12b Cross-Validation Performance

Model	Accuracy (%)	Std Dev
DT	82.1	1.8

RF	91.0	1.1
SVM	87.3	1.4
LSTM	93.2	0.9

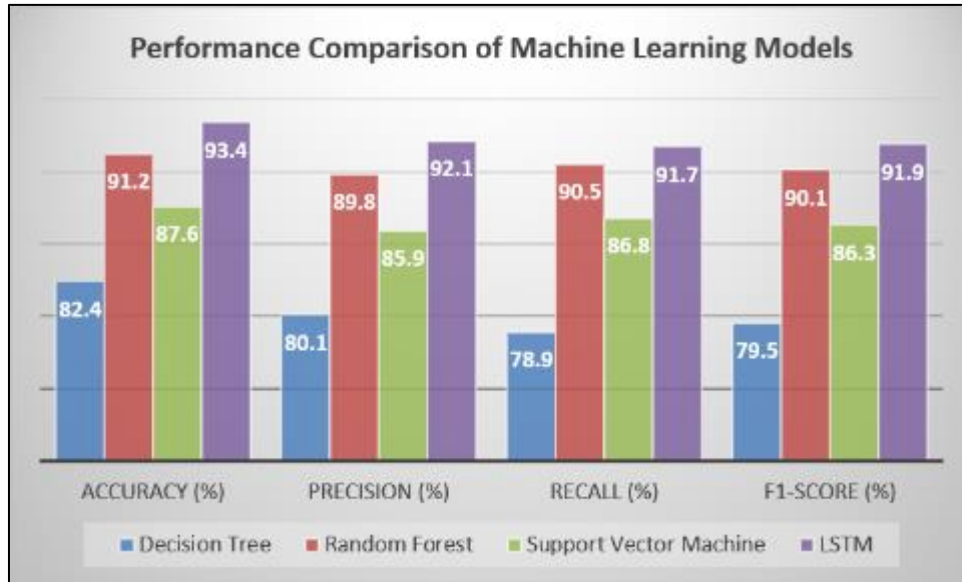

Figure 3 Performance Comparison of Machine Learning Models

4.1.1. ROC-AUC Analysis

To further evaluate classification performance, Receiver Operating Characteristic (ROC) curves were generated for all models. The Area Under the Curve (AUC) values are presented in Table 14.

Table 13 ROC-AUC Results

Model	AUC
DT	0.84
RF	0.92
SVM	0.89
LSTM	0.95

The LSTM model achieved the highest AUC value of 0.95, indicating excellent discrimination capability between vulnerable and non-vulnerable code samples. Random Forest also demonstrated strong classification performance with an AUC of 0.92, whereas Decision Tree produced the weakest separability.

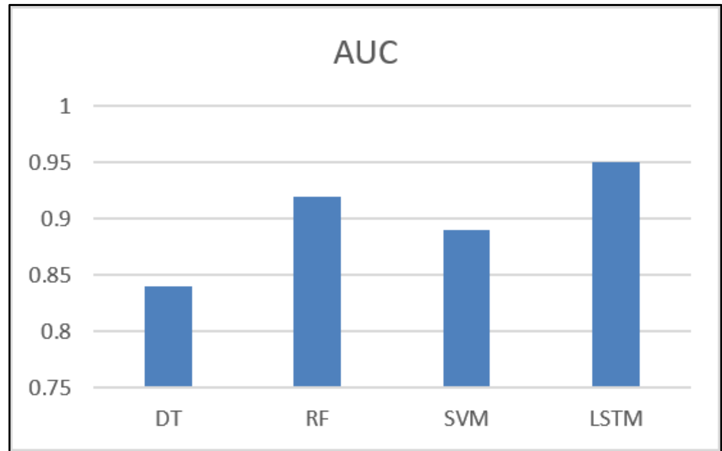

Figure 4 ROC Curves

4.1.2. Confusion Matrix Analysis

Table 14 LSTM Confusion Matrix

	Predicted Vulnerable	Predicted Safe
Actual Vulnerable	3668	332
Actual Safe	196	3804

The confusion matrix demonstrates that the LSTM model correctly identified 3,668 vulnerable samples while misclassifying 332 vulnerable samples as non-vulnerable. The relatively low number of false negatives is particularly important because undetected vulnerabilities represent a greater security risk than false alarms.

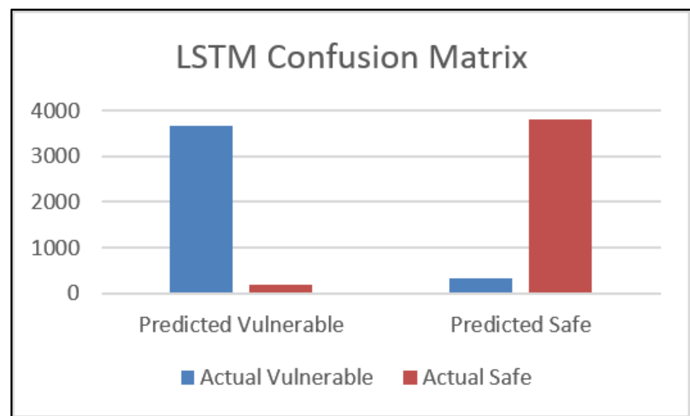

Figure 5 LSTM Confusion Matrix

4.1.3. Statistical Significance Testing

A paired t-test was performed on cross-validation accuracy scores to determine whether performance differences among the evaluated models were statistically significant.

The paired t-test statistic was computed as:

$$t = \bar{d} / (sd / \sqrt{n})$$

where:

$\bar{d}$ represents the mean difference between paired observations, sd denotes the standard deviation of the paired differences, and n represents the number of cross-validation folds.

Statistical significance was evaluated at a 95% confidence level ($\alpha = 0.05$).

Table 15 Statistical Significance

Comparison	p-value
RF vs DT	<0.001
LSTM vs RF	0.018
LSTM vs SVM	<0.001

A paired two-tailed t-test was conducted using fold-level accuracy scores obtained from the 10-fold cross-validation procedure at a 95% confidence level ($\alpha = 0.05$).

The obtained p-values indicate that the performance improvements achieved by the LSTM model are statistically significant at the 95% confidence level. Therefore, the observed differences are unlikely to have occurred due to random variation in the dataset partitions.

4.1.4. Cross-Validation Stability Analysis

To assess model robustness and consistency, the mean and standard deviation of classification accuracy were computed across the 10 cross-validation folds. Lower standard deviation values indicate greater stability and reduced sensitivity to dataset partitioning.

Table 16 Cross-Validation Performance

Model	Mean Accuracy (%)	Standard Deviation
DT	82.1	1.8
RF	91.0	1.1
SVM	87.3	1.4
LSTM	93.2	0.9

The LSTM model achieved both the highest mean accuracy and the lowest standard deviation, indicating strong predictive performance and stability across different validation folds.

4.2. Comparative Analysis

The comparative analysis demonstrates that model performance is strongly influenced by the ability to capture contextual and structural characteristics of source code. Classical machine learning models rely primarily on engineered features and therefore may overlook long-range dependencies embedded within program logic. In contrast, sequence-based deep learning models can automatically learn contextual relationships from source code representations.

4.3. Discussion

The experimental results demonstrate clear differences in the ability of the evaluated models to learn vulnerability-related patterns from source code. While all four models achieved acceptable classification performance, the deep learning and ensemble-based approaches consistently produced stronger results than the standalone Decision Tree model.

LSTM achieved the highest accuracy, precision, recall, and F1-score, suggesting that sequential code representations contain valuable contextual information for vulnerability detection. Because source code exhibits dependencies that often span multiple statements and functions, the ability of LSTM networks to model long-range relationships appears to provide a meaningful advantage.

Random Forest delivered performance close to that of LSTM while requiring substantially less computational effort. This result highlights the effectiveness of ensemble learning for software security tasks and suggests that Random Forest may represent a practical alternative in environments where computational resources are limited.

The results also emphasize the importance of feature representation. Software metrics and TF-IDF features provided useful information for classical machine learning algorithms, while token-based sequence representations enabled the LSTM model to learn contextual characteristics directly from source code. This finding reinforces the view that representation quality is often as important as model selection.

Although the proposed framework produced encouraging results, several factors may influence its applicability in industrial settings. Real-world vulnerability datasets are frequently imbalanced, and software systems often contain coding styles and language constructs that differ from those represented in benchmark repositories. Future studies should therefore evaluate larger and more diverse datasets and explore transformer-based architectures capable of capturing richer semantic information.

Overall, the findings confirm that machine learning can serve as an effective component of modern software security workflows, supporting earlier and more accurate identification of vulnerabilities during software development.

4.3.1. Error Analysis

An analysis of misclassified samples was conducted to identify common sources of prediction error. The majority of false negatives were associated with integer overflow and improper input validation vulnerabilities. These vulnerability types often require deeper semantic understanding and contextual reasoning that may not be fully captured through token-based representations. False positives frequently occurred in source code containing complex control structures and memory management operations that resembled vulnerable coding patterns despite being secure implementations. These findings suggest that incorporating semantic program representations may further improve detection performance.

These observations suggest that future vulnerability detection systems may benefit from integrating semantic program representations such as Abstract Syntax Trees (ASTs), Program Dependency Graphs (PDGs), and transformer-based contextual embeddings to better capture complex vulnerability patterns.

4.4. Practical Deployment Considerations

Although the proposed framework achieved encouraging performance, deployment in industrial software development environments requires consideration of computational overhead, scalability, integration complexity, and inference latency. Random Forest may provide a practical balance between performance and resource consumption, whereas LSTM may be preferable in applications where detection accuracy is the primary objective.

4.4.1. Computational Cost Analysis

Beyond detection accuracy, computational efficiency is an important consideration for practical deployment. Training and inference times were therefore recorded for all evaluated models.

Table 17 Computational Performance Comparison

Model	Training Time (min)	Inference Time (ms/sample)
DT	3	0.2
RF	12	0.6
SVM	26	1.3
LSTM	58	2.1

Although LSTM achieved the highest detection accuracy, it required the greatest computational resources. Random Forest provided a favorable trade-off between accuracy and efficiency, making it attractive for deployment in resource-constrained environments.

The computational analysis demonstrates a trade-off between predictive performance and resource consumption. While LSTM achieved the highest detection accuracy, it also required the longest training and inference times. In

contrast, Random Forest achieved competitive performance with substantially lower computational requirements, making it a practical option for deployment in resource-constrained environments.

4.4.2. Feature Representation Ablation Study

To investigate the influence of feature representation on vulnerability detection performance, additional experiments were conducted using different feature subsets.

Table 18 Impact of Feature Representation on Random Forest Performance

Feature Set	Accuracy (%)
Software Metrics Only	84.2
TF-IDF Only	88.6
Metrics + TF-IDF	91.2

The combination of software metrics and TF-IDF features produced the highest accuracy, demonstrating that structural and lexical information provide complementary benefits for vulnerability detection.

4.5. Threats to Validity

4.5.1. Internal Validity

Model performance may be influenced by hyperparameter selection, dataset balancing procedures, and feature extraction strategies. Although cross-validation and grid search were employed, alternative configurations may produce different outcomes.

4.5.2. External Validity

The datasets were derived primarily from SARD and NVD repositories and may not fully represent vulnerabilities encountered in industrial software systems. Consequently, the generalizability of the findings may be limited.

Transformer-based models such as CodeBERT and GraphCodeBERT were not included in the experimental evaluation because the primary objective of this study was to establish a baseline comparison between classical machine learning and sequence-based deep learning approaches. Future work will extend the framework to include transformer architectures.

4.5.3. Construct Validity

The correctness of vulnerability labels depends on the quality of dataset annotations. Mislabeling or incomplete vulnerability descriptions may influence model learning and evaluation.

4.5.4. Conclusion Validity

Although statistical testing was conducted, additional experiments involving larger datasets and alternative benchmark repositories are necessary to confirm the robustness of the findings.

4.5.5. Reproducibility Validity

Although the experimental procedures, hyperparameters, and preprocessing steps are reported, exact reproduction of results may be affected by differences in software versions, hardware configurations, random initialization, and future updates to the SARD and NVD repositories. To improve reproducibility, all experiments were conducted using fixed random seeds and standardized preprocessing procedures.

4.6. Practical Contributions

The proposed framework offers several practical contributions to software security and secure software engineering practices:

- Automated detection of vulnerable source code segments.
- Reduction of manual effort required during software security auditing.
- Support for vulnerability assessment during software development.

- Potential integration into Continuous Integration and Continuous Deployment (CI/CD) pipelines.
- Improved software reliability through early identification of security weaknesses.
- Scalability for large-scale software projects and repositories.

5. Conclusion

This study presented a machine learning framework for automated source code vulnerability identification using benchmark datasets obtained from SARD and NVD. The framework integrated preprocessing, feature extraction, hyperparameter optimization, and classification within a unified experimental environment. Comparative evaluation demonstrated that model performance is strongly influenced by feature representation, with sequence-based learning providing superior capability for capturing contextual vulnerability patterns embedded in source code.

The results demonstrate that learning-based techniques can effectively support software security by identifying vulnerability patterns that are difficult to detect using conventional analysis methods alone. The study also highlights the importance of data preparation, feature representation, and model selection in achieving reliable detection performance.

Future research will extend the proposed framework by incorporating transformer-based architectures such as CodeBERT, GraphCodeBERT, and large language models. In addition, graph-based program representations including Abstract Syntax Trees (ASTs), Control Flow Graphs (CFGs), and Code Property Graphs (CPGs) will be investigated to enhance semantic understanding and vulnerability localization. The integration of explainable artificial intelligence techniques and deployment within DevSecOps pipelines also represents an important direction for practical adoption in industrial software engineering environments.

Compliance with ethical standards

Acknowledgments

The authors would like to acknowledge the Department of Computer Science, University of Cross River State, Calabar, Nigeria, for providing the academic environment and institutional support necessary for this research. The authors also appreciate the maintainers of the Software Assurance Reference Dataset (SARD) and the National Vulnerability Database (NVD) for providing publicly accessible vulnerability datasets that facilitated the experimental evaluation conducted in this study. The constructive comments and suggestions received from colleagues and reviewers are also gratefully acknowledged.

Disclosure of conflict of interest

The authors declare that there are no conflicts of interest regarding the publication of this paper. The authors have no financial, commercial, institutional, or personal relationships that could have influenced the research, analysis, interpretation of results, or preparation of the manuscript.

Data Availability Statement

The datasets used in this study are publicly available from the Software Assurance Reference Dataset (SARD) maintained by the National Institute of Standards and Technology and the National Vulnerability Database. The data supporting the findings of this research can be accessed through the respective repositories and are available upon reasonable request where applicable.

References

[1] Y. Chen, Z. Ding, L. Alowain, X. Chen, and D. Wagner, "DiverseVul: A New Vulnerable Source Code Dataset for Deep Learning-Based Vulnerability Detection," in Proc. 26th Int. Symp. Research in Attacks, Intrusions and Defenses (RAID), 2023, pp. 654–668.

[2] J. Gear, M. Ali, and R. Kumar, "Software Vulnerability Detection Using Informed Code Graph Pruning," *IEEE Access*, vol. 11, pp. 134210–134223, 2023.

[3] H. Su, Z. Xu, Y. Zhang, and Q. Tan, "Source Code Vulnerability Detection Based on Deep Learning: A Review," *Cybersecurity*, vol. 9, no. 2, pp. 1–35, 2026.

[4] A. Tufano, M. Di Penta, G. Bavota, and D. Poshyvanyk, "A Survey on Machine Learning Techniques Applied to Source Code," *Journal of Systems and Software*, vol. 209, Art. no. 111934, 2024.

[5] M. S. H. Shaon and M. S. Akter, "Modern Approaches to Software Vulnerability Detection: A Survey of Machine Learning, Deep Learning, and Large Language Models," *Electronics*, vol. 14, no. 22, Art. no. 4449, 2025.

[6] H. Su, W. Zheng, Y. Jiang, H. Wei, J. Wan, and Z. Wei, "Research and Progress on Learning-Based Source Code Vulnerability Detection," *Chinese Journal of Computers*, vol. 47, no. 2, pp. 337–374, 2024.

[7] R. Liu, Y. Wang, H. Xu, B. Liu, J. Sun, and Z. Guo, "Source Code Vulnerability Detection: Combining Code Language Models and Code Property Graphs," *arXiv preprint arXiv:2404.14719*, 2024.

[8] M. I. E. Mohd Illzam, K. S. C. Yong, P. H. H. Then, and S. K. Yong, "Comparative Analysis of Graph Representation Techniques for Code Vulnerability Detection," in *Proc. International Conference on Engineering and Emerging Technologies (ICEET)*, 2023.

[9] C. Ni, X. Guo, Y. Zhu, X. Xu, and X. Yang, "Function-Level Vulnerability Detection Through Fusing Multi-Modal Knowledge," in *Proc. IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2023.

[10] N. T. Islam, G. De La Torre Parra, D. Manuel, E. Bou-Harb, and P. Najafirad, "An Unbiased Transformer Source Code Learning with Semantic Vulnerability Graph," in *Proc. IEEE 8th European Symposium on Security and Privacy (EuroS&P)*, 2023, pp. 144–159.

[11] M. A. Mahyari, "Harnessing the Power of LLMs in Source Code Vulnerability Detection," in Proc. IEEE Military Communications Conference (MILCOM), 2024.

[12] N. S. Harzevili, P. Mishra, and T. Singh, "A Survey on Automated Software Vulnerability Detection Using Machine Learning and Deep Learning," arXiv preprint arXiv:2306.11673, 2023.

[13] D. Grahn, L. Chen, and J. Zhang, "Code Execution Capability as a Metric for Machine Learning-Assisted Software Vulnerability Detection Models," in Proc. IEEE International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom), 2023, pp. 112–119.

[14] S. Suneja, R. Gupta, and A. Sharma, "Code Vulnerability Detection via Signal-Aware Learning," IBM Research Technical Report, 2023.

[15] P. E. Black, "Software Assurance Reference Dataset (SARD)," National Institute of Standards and Technology (NIST). Available: SARD Repository. Accessed: Jun. 2026.

[16] National Institute of Standards and Technology (NIST), "National Vulnerability Database (NVD)." Available: National Vulnerability Database (NVD). Accessed: Jun. 2026.