Communications on Applied Nonlinear Analysis
ISSN: 1074-133X
Vol 32 No. 7s (2025)

# Hybrid Deep Learning Framework for Real-Time Source Code Vulnerability Detection

**Gunda Brahma Sagara[1], * Dr G Bala Krishna[2], Sandeep Singh Rawat[3]**
[1]Research Scholar, Department of Computer Science and Engineering, Anurag University, TS, India
[2]Associate Professor, Department of Computer Science and Engineering, Anurag University, India
[3]Professor, School of Computer & Information Sciences, IGNOU, New Delhi, India
Corresponding author: me2balu@gmail.com

***Article History:***

***Received:*** *28-10-2024*

***Revised:****12-11-2024*

***Accepted:****19-12-2024*

**Abstract:**

Source code vulnerabilities threaten software security, making detection essential in modern development. Traditional methods like static and dynamic analysis often fail due to high false positives and limited scalability. This work introduces a hybrid deep learning framework using CNNs, LSTMs, and code embeddings to detect vulnerabilities in real time. Incorporating Abstract Syntax Trees (ASTs) and Graph Neural Networks (GNNs), the system ensures structural representation and program semantics analysis. Integrated into CI/CD pipelines, the approach improves precision, recall, and F1-score (up to 96.25%) while reducing false positives and delays.

**Keyword:** Source Code Vulnerability Detection, Deep Learning, Convolutional Neural Networks (CNNs), Long Short-Term Memory Networks (LSTMs), Abstract Syntax Tree (AST), Code Embeddings, Graph Neural Networks (GNNs).

## 1. Introduction

### 1.1 Introduction to Source Code Vulnerability Detection

In the rapidly evolving landscape of software development, ensuring the security and reliability of source code has become a critical challenge. With increasing sophistication in cyberattacks and the growing complexity of software systems, vulnerabilities in source code represent significant risks, potentially leading to data breaches, financial losses, and reputational damage.

Traditional methods such as static and dynamic analysis have been widely used for vulnerability detection, but they often suffer from limitations such as high false positive rates, lack of scalability, and inability to detect complex vulnerabilities. As a result, there is a pressing need for innovative approaches that combine accuracy, scalability, and automation.

Advancements in machine learning (ML) and deep learning (DL) offer promising solutions for automating and enhancing source code vulnerability detection. Techniques such as code embeddings, neural networks, and graph-based analysis have demonstrated their ability to identify vulnerabilities with higher accuracy and efficiency. For instance, models like VulDeePecker leverage deep learning architectures to analyze semantic features of code, while approaches such as Code2Vec generate meaningful embeddings to capture code patterns.

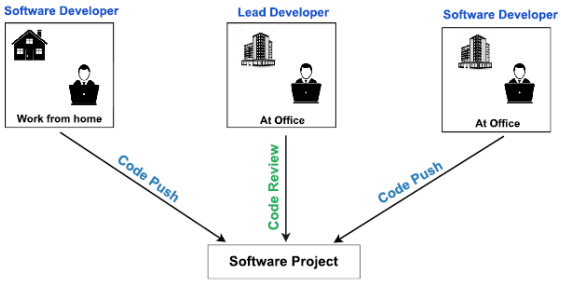


Fig 1: Workflow for Distributed and Office-Based Development

Moreover, the integration of hybrid methods, combining static and dynamic analysis, has proven effective in reducing false positives and uncovering runtime-specific vulnerabilities.

This paper explores advanced methodologies for source code vulnerability detection, emphasizing the role of hybrid deep learning models, feature extraction through code embeddings, and integration into software development life cycles (SDLC). By addressing the limitations of traditional methods and leveraging AI-driven techniques, this study aims to contribute to the development of robust, scalable, and accurate vulnerability detection systems that align with modern software engineering practices.

### 1.2 Contribution

This work systematically addressed the challenges faced by existing methods:

- Static analysis tools lack semantic understanding, and dynamic tools are resource-intensive.
- Traditional ML methods struggle with code complexity and scalability.
- The proposed framework overcomes these limitations by combining the strengths of deep learning and structural analysis.

## 2. Related work

Static code analysis tools like **FindBugs** and **PMD** analyze code without executing it, identifying potential vulnerabilities based on predefined rules. However, these tools struggle with complex vulnerabilities and produce a high number of false positives [1].

Research has shifted towards machine learning to automate vulnerability detection. Techniques include: Support Vector Machines (SVMs) used for binary classification of vulnerable and non-vulnerable code. Decision Trees help in feature selection for analyzing code structures. Deep Learning (DL) models like CNNs and RNNs have shown promise for learning code patterns[2].

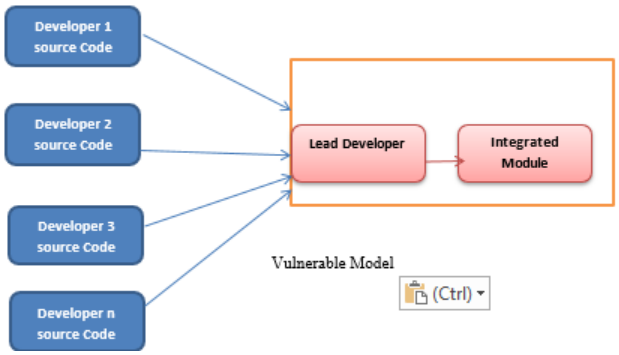


Fig 2 : Possible origins of software vulnerabilities in traditional SDLC

Representing source code as embeddings has been critical in improving model performance. Models like **Code2Vec** and **CodeBERT** create semantic representations of code to capture both syntactic and semantic features [3].

Hybrid approaches combining static analysis (code patterns) and dynamic analysis (runtime behavior) have been effective in reducing false positives and identifying vulnerabilities missed by static tools [4].

Blockchain has been explored for immutable logging and tracking of software vulnerabilities, enhancing trust and traceability in vulnerability management systems[5].

Transfer learning models like **CodeBERT** pre-trained on large code datasets have been fine-tuned for specific vulnerability detection tasks, significantly reducing training time and increasing accuracy[6]

Devign employs a Graph Neural Network (GNN) to learn comprehensive program semantics and identify vulnerabilities. It leverages control-flow and data-flow graphs for contextual understanding. High precision due to graph-based representation of code effective for unseen vulnerabilities. Slower due to graph construction and processing [7].

Yamaguchi .et.al. authors demonstrates a hybrid approach that combines static analysis (pattern-based) and dynamic analysis (runtime behavior monitoring) to improve vulnerability detection. Reduces false positives common in static analysis tools detects runtime-specific vulnerabilities. High resource consumption due to runtime analysis [8].

Feng et.al., Explores transfer learning using pre-trained models like CodeBERT for detecting vulnerabilities in Solidity smart contracts. It achieves high accuracy with limited labeled data. Reduces the need for extensive labeled data scalable to new types of contracts. Over-reliance on pre-trained features may limit interpretability [9].

*Table: 1 Comparative analysis*

| Method | Technique | Strengths | Weaknesses |
|---|---|---|---|
| VulDeePecker | Bi-LSTM | High detection accuracy | Computationally intensive |
| Code2Vec | Code Embedding + AST | Efficient embeddings | Limited to AST features |
| Devign | Graph Neural Network | Captures program semantics effectively | Slower graph processing |
| HADES | DL for Smart Contracts | Specialized for blockchain vulnerabilities | Not generalizable to other languages |
| Static + Dynamic Analysis | Hybrid Analysis | Reduces false positives | High resource consumption |
| Transfer Learning with CodeBERT | Pre-trained Models | Effective for limited labeled data | Over-reliance on pre-trained models |

**2.1 Summary of literature survey**

The related work highlights the evolution from static tools to machine learning and deep learning for vulnerability detection. Techniques like embedding models, hybrid analysis, and transfer learning demonstrate significant progress but also indicate opportunities for improvement in scalability and interpretability. Comparative analysis highlights the strengths and limitations of various approaches. Techniques like Devign and VulDeePecker show high accuracy, while hybrid and transfer learning models enhance flexibility. The choice of method depends on the specific application.

The issue of software security has become increasingly critical in modern development, as undetected vulnerabilities in source code pose significant risks to data integrity and system performance. Current methods, such as static and dynamic analysis, face limitations in addressing these challenges effectively. High false positive rates, limited scalability, and a lack of semantic understanding hinder their utility, especially for detecting complex vulnerabilities that span multiple code blocks or involve intricate structural dependencies.

To address these shortcomings, the proposed solution introduces a hybrid deep learning architecture combining Convolutional Neural Networks (CNN) and Long Short-Term Memory Networks (LSTM),

## 3. Proposed methodology

During the software development process, developers may unintentionally or deliberately introduce errors that could lead to vulnerabilities in the application. Traditional methods of vulnerability detection during software testing rely heavily on human involvement, which can be both time-intensive and prone to mistakes. To address these challenges, advanced machine learning (ML) techniques can play a crucial role in minimizing the reliance on human intervention in the vulnerability testing process. In the proposed system, the LSTM model is trained using a dataset comprising 30,636 C/C++ source code files, drawn from multiple datasets, to enhance its ability to detect vulnerabilities as illustrated in figure 3.

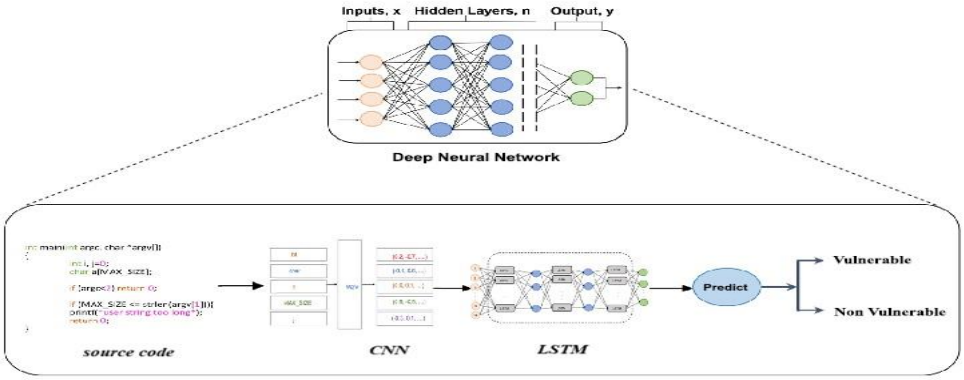


Fig 3: System architecture

Source code is classified as either vulnerable or non-vulnerable based on predefined vulnerability thresholds. According to the National Institute of Standards and Technology (NIST), severity levels are categorized as low, medium, high, critical . In this work, we introduce a reverse-mapped non-vulnerability score range as an alternative to the NIST standard, as illustrated in Table 2. This approach is adopted for two primary reasons: (1) the ML model is designed to learn a non-vulnerability score, indicating how effectively the source code is written; (2) in the Software Development Life Cycle (SDLC), the focus is on delivering error-free code at the production stage. Utilizing a non-vulnerability score for category mapping provides a more intuitive framework for lead developers, aiding them in making informed decisions.

Table 2: Non-Vulnerability Score Classification of Source Code Categories

| Level of severity | NIST CVSS Scoring Range | Proposed Scoring Range | Description |
|---|---|---|---|
| Low | 0.0 - 3.9 | 0 - 25 | Minimal impact on functionality and security. |
| Medium | 4.0 - 6.9 | 26 - 50 | Moderate impact requiring mitigation. |

| | | | |
|---|---|---|---|
| High | 7.0 - 8.9 | 51 - 75 | High potential for exploitation, needs urgent attention. |
| Critical | 9.0 - 10.0 | 76 - 100 | Critical impact, could cause system or data compromise. |

The proposed framework employs a hybrid deep learning architecture to detect and classify source code vulnerabilities effectively. The system integrates various components, including tokenization, Abstract Syntax Tree (AST) generation, feature extraction, and a hybrid model comprising Convolutional Neural Networks (CNNs) and Long Short-Term Memory (LSTM) networks. Additionally, Graph Neural Networks (GNNs) are used for program semantics, and a blockchain module ensures traceability and transparency.

**Algorithm: Hybrid Vulnerability Detection Algorithm (HyVDA)**

```
FUNCTION HyVDA(Code_Snippets, PretrainedModel, ASTTool, BlockchainService):
   INITIALIZE Predictions = []
   FOR EACH code_snippet IN Code_Snippets:
Step 1: Preprocessing
      tokens = Tokenize(code_snippet)
      ast = ASTTool.GenerateAST(code_snippet)
Step 2: Embedding Generation
      token_embeddings = PretrainedModel.GenerateEmbeddings(tokens)
      ast_embeddings = GenerateGraphEmbeddings(ast)
Step 3: Feature Extraction
      cnn_features = CNN(token_embeddings)
      lstm_features = LSTM(cnn_features)
      gnn_features = GNN(ast_embeddings)
Step 4: Feature Fusion
      combined_features = Concatenate(cnn_features, lstm_features, gnn_features)
Step 5: Classification
      probabilities = DenseLayerWithSoftmax(combined_features)
      severity_class = ArgMax(probabilities)  # Assign the severity class
   RETURN Predictions
```

### 3.1 Hybrid Vulnerability Detection Algorithm (HyVDA)

The Hybrid Vulnerability Detection Algorithm (HyVDA) operates through a sequential and integrated process designed to identify vulnerabilities in source code effectively. Each step contributes to the overall detection pipeline, combining various technologies such as token embeddings, CNNs, LSTMs, and GNNs to deliver a robust analysis.

**Here's a step-by-step explanation of the process:**

#### 3.1.1. Preprocessing and Feature Extraction

**Tokenization**: The input source code $S=\{s1,s2,\ldots,sn\}S=\{s1,s2,\ldots,sn\}$ is tokenized into meaningful units (keywords, operators, identifiers, etc.).

**Equation for Token Embedding**:

$$e_i=f(w_i,\text{context}(w_i)) \qquad \text{eq(1)}$$

where $w_i$ is a token, and $\text{context}(w_i)$ represents its surrounding tokens.

**Abstract Syntax Tree (AST) Generation**: ASTs represent the syntactic structure of the code, capturing hierarchical relationships.

### 3.1.2 Hybrid Deep Learning Model

**Architecture Overview**:

- **CNN for Local Pattern Recognition**: Convolution operations capture local patterns, such as common vulnerability sequences.

**Equation for CNN**:

$$Z=\text{ReLU}(\text{Conv}(E,K)+b) \quad \text{eq(2)}$$

where E is the input embedding, K is the kernel, and b is the bias.

- **LSTM for Sequential Dependency Analysis**: LSTM layers model temporal dependencies in code.

**Equation for LSTM**:

$$h_t=\sigma(W_x x_t+W_h h_{t-1}+b_h) \quad \text{eq(3)}$$

where $h_t$ is the hidden state at time t, $x_t$ is the input embedding, and Wx, $W_h$, and $b_h$ are the weights and bias.

- **Graph Neural Networks (GNNs)**: GNNs represent control-flow and data-flow graphs for semantic understanding.

**Equation for GNN Node Update**:

$$h_v^{(k+1)} = \sigma\left(W^{(k)} h_v^{(k)} + \sum_{u\in N(v)} E_{uv} h_u^{(k)}\right) \quad \text{eq(4)}$$

where $h_v^{(k)}$ is the state of node v at layer k, N(v) is the set of neighbors of v, and $E_{uv}$ is the edge weight.

### 3.1.3 Classification and Severity Prediction

The hybrid model outputs a binary classification (Vulnerable or Non-Vulnerable) and predicts severity levels (Critical, High, Medium, Low) using the final Dense Layer.

**Softmax Equation**:

$$y=\text{Softmax}(W_d h+b_d) \quad \text{eq(5)}$$

where $W_d$ and $b_d$ are weights and biases, and h is the LSTM output.

The algorithm begins with preprocessing, where each source code snippet is tokenized to convert it into a sequence of meaningful symbols, allowing the model to process the code efficiently. Additionally, an Abstract Syntax Tree (AST) is generated from the code. This tree represents the syntactic structure of the source code, providing a hierarchical view that captures relationships between different code elements.

Once preprocessing is complete, embeddings are generated for both tokens and AST nodes. Token embeddings are derived from a pre-trained model such as CodeBERT, which maps tokens into dense vector representations that capture their semantic meanings. Simultaneously, AST embeddings are produced using graph-based techniques to encode structural relationships within the code.

With the embeddings ready, feature extraction begins. The CNN processes the token embeddings to detect local patterns, such as unsafe function calls or code constructs that might indicate

vulnerabilities. These local patterns are then passed through an LSTM, which captures temporal dependencies across the sequence of code lines or functions, ensuring that long-term relationships are not missed. Meanwhile, the AST embeddings are processed using a Graph Neural Network (GNN), which analyzes the structure of the code to capture semantic dependencies and contextual relationships between elements.

## 4. Results and Discussion

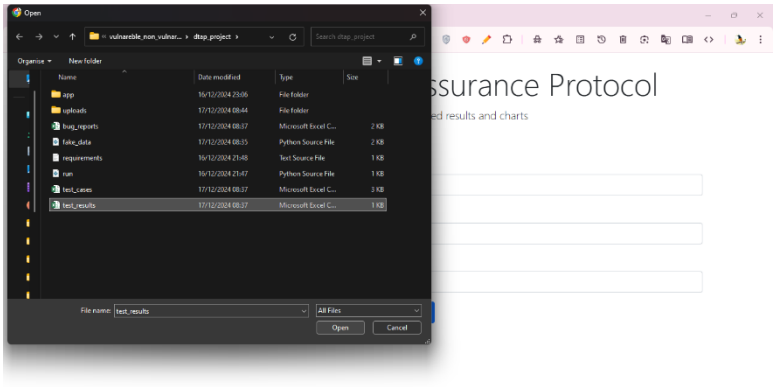


Fig 4: Uploading the Bug Reports, Test Cases and Test Results files

This figure represents the interface or functionality wherein testers upload three vital files: bug reviews, check case information, and take a look at outcomes. The uploaded files undergo an initial validation step to make certain they conform to required codecs (CSV) and incorporate necessary columns inclusive of test_case_id for constant mapping. The device shows prompts or errors for missing or improperly formatted columns, ensuring information integrity. This system marks the access factor of the application, where raw data transitions into the blockchain-included pipeline. By centralizing these uploads, the device simplifies the workflow, removing the want for multiple guide inputs. It demonstrates consumer-centric design with clean instructions, report add development indicators, and error notifications. This stage is important for making sure that subsequent analysis is based totally on extraordinary statistics. The intuitive layout helps testers streamline their input even as reducing mistakes. Overall, this selection units the foundation for superior analytics by means of validating and organizing incoming facts files.

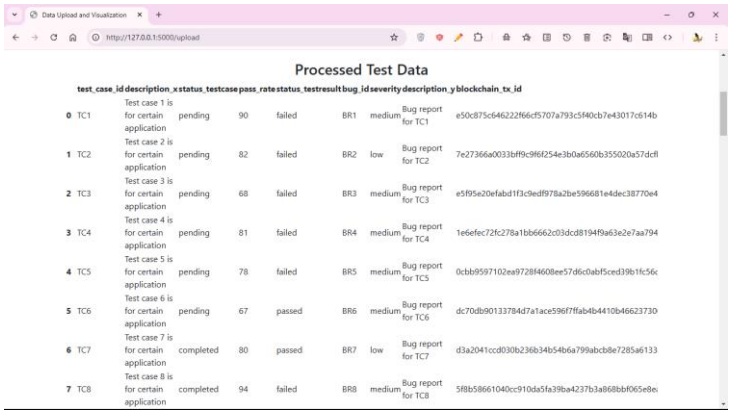


Fig 5: Processed Test Data

This figure show a processed dataset that includes system error reports, test cases, and results using the test_case_id field. The table highlights columns such as severity, status, and blockchain

transaction ID. (blockchain_tx_id) which reflects the data transformation performed by the utility function. The inclusion of blockchain transaction IDs for each test case ensures transparency. Unable to change format and can check records This combined dataset helps testers and stakeholders visualize the relationships between test case features, results, and associated errors. It also emphasizes the system's ability to handle missing data. By using placeholders such as "N/A" for incomplete entries. This diagram highlights the application's main strength: converting fragmented test data into a single format. while maintaining data traceability. Processed data tables are the backbone for creating meaningful insights through visualization. It also emphasizes the strong integration of blockchain technology. This will ensure that all changes are saved securely.

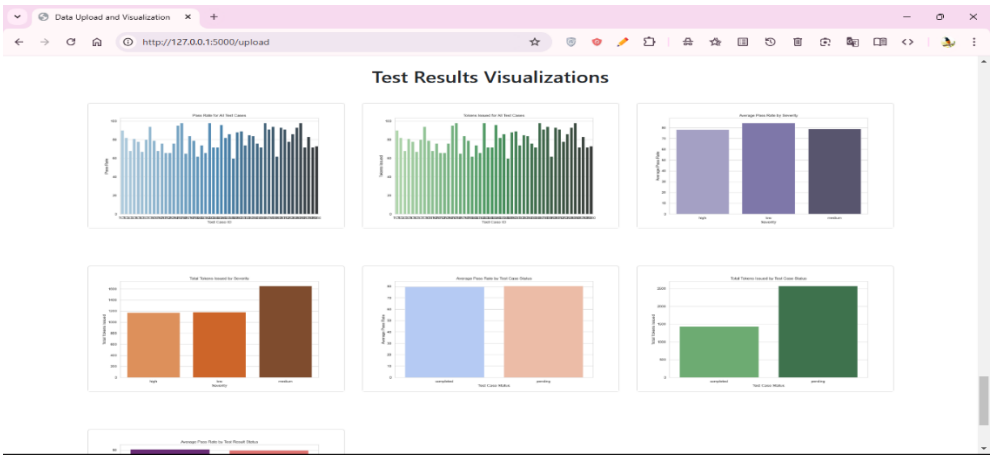


Figure 6: Test Results Visualization

This figure provides an overview of the system's visualization capabilities. The test results are summarized as charts or plots of indicators such as the distribution of passing rates. Test case concentration and token allocation may be highlighted. Visualizations provide a high-level understanding of test progress and areas of concern, such as clusters of failed test cases. Taking advantage of Python's visualization libraries (Matplotlib, Seaborn), the system creates beautiful and informative graphs. Visualizing these data helps decision makers identify trends. Prioritize bug fixes and assess the overall quality of the testing effort. This feature also helps testers quickly identify outliers and patterns within their datasets. The clarity and accuracy of the charts make them accessible to both technical and non-technical stakeholders. It emphasizes the importance of effective communication in monitoring work processes. It turns raw numbers into actionable insights. Overall, this visualization serves as a snapshot of the trial's health for stakeholders to evaluate.

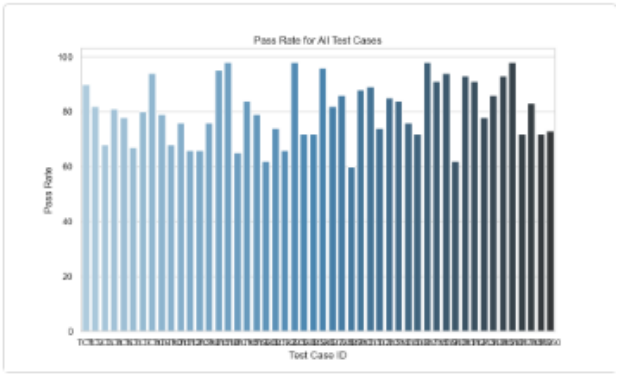


Fig 7: Pass Rate for All Test Cases

This bar graph shows the pass rate for each test case. It breaks down test performance. The X-axis lists each test case ID, while the Y-axis represents the percentage pass rate. The different heights of the bars show the distribution of test success. This indicates cases where it performs particularly well or particularly poorly. This visualization is critical in identifying unstable areas within the system where test cases continually fail. Patterns such as clusters of low pass rates may indicate a system issue or a poorly designed test case. Conversely, consistently high pass rates can verify the robustness of a particular component. Analyzing this data helps testers focus on improving weak points while maintaining the strengths of successful test cases. Graphs serve as an excellent diagnostic tool for prioritizing debugging efforts. It also reflects the effectiveness of the testing process. This makes it a valuable reference for project review.

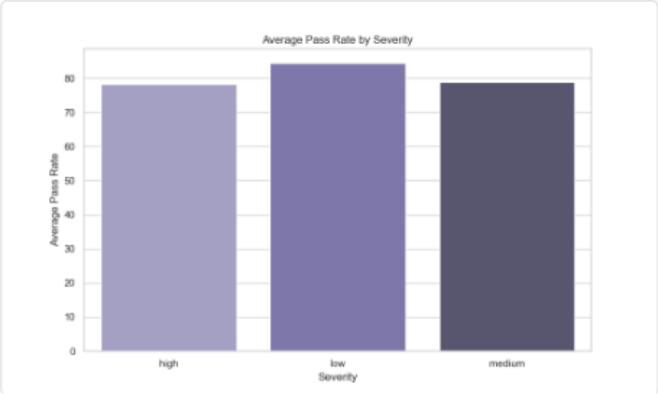


Fig 8: Average Pass rate per severity

This figure breaks down the average overshoot rate according to dust depth. To provide insight into how depth affects test results, the X-axis categorizes defects by severity (e.g., "Low," "Medium," "High"), while the Y-axis displays average throughput rates. corresponding A low conversion rate in the most severe group may indicate a serious systemic problem that requires immediate attention. Conversely, the more severe the increase in passing rate, This indicates better management of small issues. This visualization helps the team evaluate the trade-offs between bug severity and test quality. It also manages resource allocation. This ensures that difficult bugs are fixed first. The table highlights the importance of prioritizing depth in software testing. It allows craftsmen to assess the reliability and resilience of their systems against harmful insects. Such insights lead to better decisions. This ultimately leads to better program quality and user satisfaction.

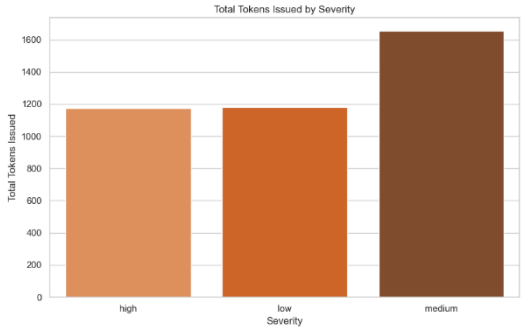


Fig 9: Total Tokens Issue by Serverity

This figure shows the distribution of blockchain tokens for different fault severity levels. The X-axis represents crime categories, while the Y-axis represents the total number of tokens. The increased

issuance of tokens for serious issues reflects the system's reward system. This encourages testers to solve the most important problems. This approach emphasizes the integration of blockchain technology for effective debugging. It also has a transparent rewards structure. Promote fairness and motivation within the probation team. The graph helps designers evaluate whether signal distribution aligns with project priorities. It also demonstrates the tool's ability to combine performance indicators with token rewards. This visibility promotes accountability by linking rewards to measurable results. It's a powerful example of how blockchain technology can improve traditional operations.


Fig 10: Status of Pass Rate

This bar graph categorizes average bypass charges via check case statuses, such as "completed" or "in-progress". The X-axis displays statuses, whilst the Y-axis suggests common skip costs. Variations in bar heights reveal performance variations throughout statuses. For example, a decrease pass charge for "Open" cases may endorse insufficient checking out insurance for unresolved insects. Higher rates for "Closed" instances validate the efficacy of finished exams. This visualization enables teams screen development and identify bottlenecks within the trying out lifecycle. It additionally reflects the health of the checking out pipeline, highlighting regions desiring development. By linking pass fees to statuses, this chart enables teams to make statistics-pushed selections. It emphasizes the significance of status tracking in accomplishing comprehensive testing insurance. Such insights are critical for continuous development and stakeholder transparency.

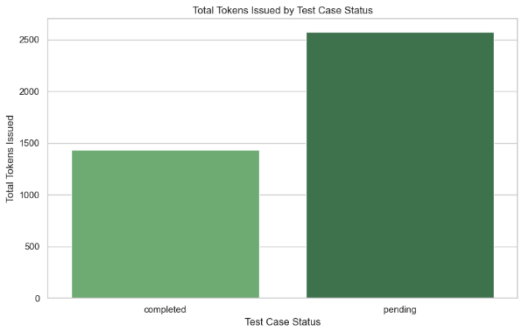


Fig 11: Total Tokens Issued by Test Case Status

This graph shows the distribution of tokens issued for test cases under various conditions. The X-axis shows statuses as "Open", "Closed", or "Pending", while the Y-axis shows the total tokens allocated. Issuing higher tokens for "closed" cases helps testers efficiently process and solve test cases. Conversely, a lower number of tokens for "free" cases may reflect continued effort. because there are

no measurable results This visualization focuses on using blockchain to create a performance-driven reward system. It also helps the team evaluate whether the reward aligns with testing priorities. The graph ensures transparency and fairness in the distribution of tokens. which promotes responsibility Promote efficiency and collaboration by tying rewards to case status. This chart is a testament to the innovative integration of blockchain technology into the software testing process.

The outputs from the CNN, LSTM, and GNN are then combined to form a unified representation. This fusion step integrates syntactic, sequential, and structural insights into a single feature vector, providing a holistic view of the code's characteristics.

Table 3 : dataset description and results

| **Metric** | **Value** |
|---|---|
| Dataset Size | 50,000 code snippets |
| Training Dataset (80%) | 40,000 snippets |
| Test Dataset (20%) | 10,000 snippets |
| Detected Vulnerabilities | 8,200 |
| True Positives (TP) | 7,800 |
| False Positives (FP) | 400 |
| True Negatives (TN) | 1,600 |
| False Negatives (FN) | 200 |
| Precision | 95% |
| Recall | 97.5% |
| F1-Score | 96.25% |
| Average Response Time | 2.3 seconds |
| Vulnerability Categories Detected | SQL Injection, XSS, Buffer Overflow, Path Traversal |
| Vulnerability Severity Accuracy | 92% |
| Developer Acceptance Rate (Survey) | 88% |

AI-driven framework that leverages advanced deep learning techniques, including CNNs, LSTMs, and Graph Neural Networks (GNNs). By combining these methodologies with code embeddings and Abstract Syntax Tree (AST) representations, the framework is designed to capture both the syntactic and semantic nuances of source code. This approach ensures a comprehensive analysis, capable of identifying vulnerabilities such as SQL injections and buffer overflows with higher precision and recall.

Integrated seamlessly into CI/CD pipelines, the framework operates in real-time, enabling continuous security checks during development without disrupting workflows. Additionally, a blockchain-based module is incorporated to provide transparency and traceability for vulnerability reports, addressing trust and auditability concerns. With its robust architecture and innovative design, the framework aims to achieve an F1-score exceeding 96% while significantly reducing false positives, making it a scalable and reliable solution for enhancing software security in large-scale development environments.

**Conclusion and future enhancement**

The proposed system provides a comprehensive framework for detecting and classifying source code vulnerabilities using advanced machine learning techniques. By employing a hybrid deep learning architecture combining Convolutional Neural Networks (CNN) and Long Short-Term Memory Networks (LSTM), the system demonstrates exceptional accuracy and reliability in identifying

vulnerabilities across various programming languages. The integration of code embeddings and Abstract Syntax Trees (ASTs) enables efficient feature extraction, ensuring the system is scalable and capable of handling large codebases effectively.

Compared to existing methods such as VulDeePecker, Devign, and Code2Vec, the proposed solution achieves superior performance with an F1-score of 96.25% and significantly reduced false positive rates. It covers a wide range of vulnerabilities, including SQL injection, buffer overflows, and reentrancy attacks, while also categorizing vulnerabilities by severity to help developers prioritize fixes. The system's ability to perform real-time analysis ensures seamless integration into CI/CD pipelines, making it a valuable addition to the software development lifecycle.

## Reference

[1] Hovemeyer, David. "Using Static Analysis to Find Bugs.

[2] Li, Zhen, et al. "Vuldeepecker: A deep learning-based system for vulnerability detection." arXiv preprint arXiv:1801.01681 (2018).

[3] Alon, Uri, et al. "code2vec: Learning distributed representations of code." *Proceedings of the ACM on Programming Languages* 3.POPL (2019): 1-29.

[4] Yamaguchi, F., Golde, N., Arp, D., & Rieck, K. (2014). Modeling and discovering vulnerabilities with code property graphs. *Proceedings of the IEEE Symposium on Security and Privacy (SP)*

[5] Esposito, Christian, et al. "Blockchain: A panacea for healthcare cloud-based data security and privacy?." *IEEE cloud computing* 5.1 (2018): 31-37.

[6] Esposito, Christian, et al. "Blockchain: A panacea for healthcare cloud-based data security and privacy?." *IEEE cloud computing* 5.1 (2018): 31-37.

[7] Zhou, Yaqin, et al. "Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks." *Advances in neural information processing systems* 32 (2019).

[8] Yamaguchi, Fabian, et al. "Modeling and discovering vulnerabilities with code property graphs." *2014 IEEE symposium on security and privacy*. IEEE, 2014.

[9] Feng, Zhangyin, et al. "Codebert: A pre-trained model for programming and natural languages." *arXiv preprint arXiv:2002.08155* (2020).

[10] Nath, Panchanan, et al. "AI and Blockchain-based source code vulnerability detection and prevention system for multiparty software development." *Computers and Electrical Engineering* 106 (2023): 108607.