IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429
Volume 5, Issue 2, May 2025
Impact Factor: 7.67

Software Vulnerability Testing using Borderline SMOTE

M. Vara Lakshmi[1], Mohammed Aqeel Yousuf[2], A Sreehan Chary[3]
Assistant Professor, Department of IT[1]
B.Tech Student, Department of IT[2,3]
Mahatma Gandhi Institute of Technology, Hyderabad, India

Abstract: *Enhancing the efficiency of software vulnerability detection has become increasingly important as software systems grow and complexity. In this study, we propose a comprehensive and innovative framework that brings together four essential techniques—Source Embedding, Feature Learning, Data Resampling, and Classification—to tackle the critical challenges of identifying vulnerabilities in source code. The process begins with Source Embedding, where Abstract Syntax Trees (ASTs) generated using the Joern tool are analysed with advanced data mining techniques to capture both the structural and semantic characteristics of the code. These embedded representations form the basis for Feature Learning, which applies machine learning and deep learning algorithms to extract meaningful insights from the AST nodes, enabling the detection of both known vulnerabilities and subtle, hidden code anomalies. Recognizing the issue of data imbalance in real-world datasets, we integrate the Borderline SMOTE algorithm to generate synthetic examples near class boundaries, helping to create a more balanced and representative dataset. With this enriched dataset, the Classification stage leverages robust models trained to accurately identify potential vulnerabilities. What sets our approach apart is the seamless integration of varied techniques, which allows us to discover intricate patterns that traditional static analysis tools often miss. We validate our framework using the Verum dataset, where it delivers outstanding results across multiple performance metrics including precision, recall, F1-score, and accuracy. The findings affirm the model's capability to deliver reliable predictions while reducing false positives and negatives. This holistic methodology not only raises the bar for source code vulnerability detection but also lays a solid foundation for building more secure and resilient software. By addressing the core aspects of code representation, feature extraction, and imbalanced learning, our study contributes significantly to the development of smarter, more adaptive security tools for modern development environments.*

Keywords: Abstract Syntax Tree (AST), Source Embedding, Feature Learning, Borderline SMOTE, Classification

I. INTRODUCTION

In today's software-driven world, the complexity and scale of codebases are expanding at an unprecedented rate. As developers strive to deliver faster and more feature-rich applications, security often becomes an afterthought, leading to hidden vulnerabilities that can be exploited by attackers. Traditional static and dynamic analysis tools, while useful, often rely on predefined rules or known vulnerability patterns, limiting their ability to detect novel or deeply embedded flaws. These limitations have prompted a growing interest in data-driven, intelligent systems that can automatically analyse source code and identify potential threats with greater accuracy and context awareness. Detecting vulnerabilities early in the development lifecycle is crucial not only for reducing risk but also for saving time, effort, and costs associated with post-deployment fixes.

This research introduces a comprehensive and intelligent framework that leverages a combination of structural analysis, deep learning, and synthetic data augmentation to detect vulnerabilities in source code more effectively. Using Abstract

Syntax Trees (ASTs) generated by the Joern tool, the system captures both the structural and semantic features of the code. These features are then processed through machine learning and deep learning models to uncover both explicit vulnerabilities and more subtle, context-dependent code anomalies. To address the common issue of data imbalance—where non-vulnerable samples vastly outnumber vulnerable ones—the Borderline SMOTE algorithm is used to generate synthetic data near decision boundaries, helping models learn more effectively. Finally, the classification module analyses the balanced, feature-rich dataset to accurately detect vulnerable code segments. The approach is validated on the Verum dataset and shows strong performance across key metrics, highlighting its potential to enhance modern software security practices with minimal manual effort.

II. LITERATURE SURVEY

The study "A Novel Approach for Software Vulnerability Detection Based on Intelligent Cognitive Computing" by Cho Do Xuan et al. (2023) introduces the EFRC Framework, which uses Code Property Graphs (CPGs) to enable semantic analysis of source code. To address class imbalance, it employs the SMOTE algorithm for generating synthetic data points, though it may introduce irrelevant code variations. The framework also uses a triplet loss function for classification, clustering vulnerable instances while separating them from non-vulnerable ones [1]. In "Data-driven Software Vulnerability Assessment and Prioritization" (2022), Triet H. M. Le and Huaming Chen explore how machine learning (ML) and deep learning (DL) techniques, such as BERT and word2vec, are used for semantic feature extraction from textual data. While these methods can analyze small datasets effectively, they face challenges with domain-specific vulnerabilities and data quality issues, limiting their generalizability [2].

In "SQVDT: A Scalable Quantitative Vulnerability Detection Technique for Source Code Security Assessment" (2021), Junaid Akram proposes detecting vulnerabilities using CVE numbers, mapping patch files to these numbers to identify vulnerabilities in source code. This method leverages a reference fingerprint index of vulnerable files for scalability and proactive vulnerability detection [3]. In "Detecting Software Vulnerabilities Using Language Models" (2023), Marijke de Vet presents VulDetect, which fine-tunes a pre-trained GPT model to detect vulnerabilities by learning contextual patterns in code. The effectiveness of this approach depends on the quality and relevance of the pre-trained model used, but it offers an automated and promising method for vulnerability detection [4].

The study "Fine-Tuning Pre-Trained CodeBERT for Code Search in Smart Contract" (2023) by Huan Jin and Qinying Li discusses how fine-tuning CodeBERT improves code search accuracy in smart contracts. By using domain-specific datasets, the approach enhances the model's performance in understanding both programming languages and natural language [5]. Finally, in "Research on Customer Churn Intelligent Prediction Model based on Borderline-SMOTE and Random Forest" (2022), L. Feng introduces a model that combines Borderline-SMOTE with Random Forest to address class imbalance in customer churn prediction. This method generates synthetic samples near the decision boundary, improving the model's predictive accuracy and robustness [6].

The surveyed studies collectively highlight the evolving methods for software vulnerability detection, combining advanced techniques from machine learning, deep learning, and cognitive computing. From utilizing graph-based structures like Code Property Graphs (CPGs) for semantic analysis to fine-tuning transformer models like GPT and CodeBERT for vulnerability detection, these approaches offer innovative ways to address issues like class imbalance and contextual understanding of code. Additionally, methods such as the use of CVE numbers for pattern detection and combining SMOTE-based resampling with machine learning models show a holistic approach to improving vulnerability detection accuracy. While challenges remain, particularly with domain-specific issues and data quality, the research demonstrates significant progress in automating and enhancing software security through intelligent, data-driven techniques.

III. METHODOLOGY

Multilayer Perceptron (MLP)

Purpose: In this project, the Multilayer Perceptron (MLP) is used to understand the relationships between different parts of the source code — such as how variables, functions, and logic blocks are connected. These connections are captured from the edges of something called an Abstract Syntax Tree (AST), which visually maps out the structure of the code. Since many vulnerabilities arise from how different parts of the code interact, rather than from individual lines, MLP helps us focus on these relationships. By learning patterns in these connections, MLP enables the system to spot vulnerabilities that are subtle and not easily visible through surface-level code analysis.

How it Works: An MLP is a type of feedforward neural network composed of multiple layers: an input layer, one or more hidden layers, and an output layer. It transforms input features through a series of weighted connections and non-linear activations. In this case, the input **x** consists of edge features from the AST.

The computation in an MLP follows this formula:

$$\mathbf{y = \sigma\,(W_2 \,.\, \sigma(\,W_1 \,.\, x + b_1\,) + b_2\,)}$$

Where:

x is the input feature vector.

$\mathbf{W_1}$, $\mathbf{W_2}$, are the weight matrices.

$\mathbf{b_1}$, $\mathbf{b_2}$ are the bias terms.

σ is the activation function (typically ReLU or Sigmoid).

Graph Convolutional Network (GCN)

Purpose: The Graph Convolutional Network (GCN) is used to capture features from the nodes of the Abstract Syntax Tree (AST), which represent individual components like functions, variables, or expressions. Unlike other models, GCN is designed to work with graph-based data, making it well-suited for processing the AST. By focusing on the nodes and their relationships, GCN helps us understand how components in the code interact, allowing the model to detect subtle vulnerabilities that may arise from these interactions.

How it Works: GCN learns by aggregating information from neighbouring nodes in the graph. Each node in the AST starts with its own feature set, representing its characteristics. GCN then updates these features by combining them with the features from neighboring nodes, essentially learning from the relationships between different components of the code. This process happens across multiple layers, refining the node representations and helping the model capture complex patterns in the code structure. By considering the context around each node, GCN improves its ability to detect vulnerabilities based on how the code components are interconnected.

The computation in an GCN follows this formula:

$$\mathbf{H^{(K+!)} = \sigma\,(\,A \,.\, H^{(K)} \,.\, W^{(K)}\,)}$$

Where:

H(k) is the feature matrix at the k-th layer.

A is the normalized adjacency matrix (structure of the graph).

W(k) is the weight matrix at the k-th layer.

Σ is the activation function (commonly ReLU).

Borderline SMOTE

Purpose: Borderline SMOTE is used to address the class imbalance problem by generating synthetic samples of the minority class, especially near the decision boundary. This helps the model focus on the more challenging areas where misclassifications are likely to occur, improving its ability to detect vulnerabilities in edge cases.

How it Works: Borderline SMOTE works by creating new synthetic data points near the border between the majority and minority classes. It identifies the instances that are closest to the decision boundary and generates new samples by interpolating between them. This helps balance the dataset and makes the model more sensitive to subtle vulnerabilities.

The formula for Borderline SMOTE is:

$$\mathbf{x_{(new)} = x_{(minority)} + \lambda \,.\, (\,x_{(near)} - x_{(minority)}\,)}$$

Where:

$\mathbf{x_{(minority)}}$ is a point from the minority class (vulnerable code).

$\mathbf{x(_{near})}$ is a nearest neighbor of the minority class point.

λ is a random value between 0 and 1, which controls the interpolation between points.

Triplet Loss

Purpose: Triplet Loss is used to optimize the model's ability to differentiate between similar and dissimilar examples. It helps the model learn a feature space where vulnerable code is distinct from non-vulnerable code, improving its classification accuracy**.**

How it Works: Triplet Loss works by comparing three inputs: an anchor (original sample), a positive (similar sample), and a negative (dissimilar sample). The goal is to minimize the distance between the anchor and positive samples while maximizing the distance between the anchor and negative samples. This ensures that the model learns to better distinguish between vulnerabilities and non-vulnerabilities.

IV. RESULT ANALYSIS

The vulnerability detection system performed effectively across its various phases, each contributing to the successful identification of vulnerabilities in source code. In the **Source Embedding** phase, the system converted the source code into Abstract Syntax Trees (ASTs), providing a structured representation of the code. This transformation was crucial as it laid the groundwork for the next phase, enabling the extraction of meaningful features that captured the relationships between different components of the code.

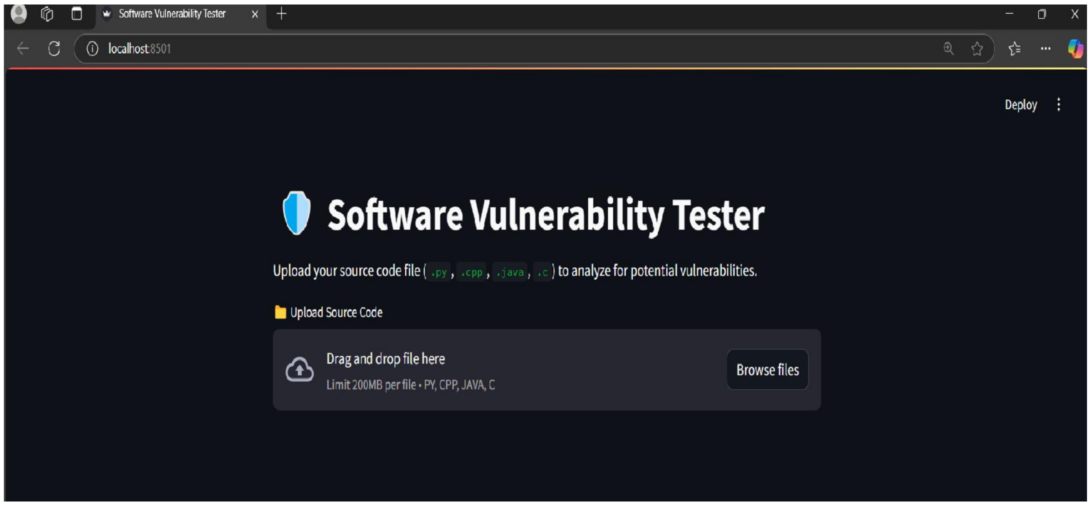
In the **Feature Extraction** phase, the combination of **Graph Convolutional Networks (GCN)** and **Multilayer Perceptron (MLP)** allowed the system to capture intricate patterns in the code. The GCN efficiently aggregated information from neighbouring nodes in the AST, helping to identify complex relationships between code components, while the MLP focused on the edges, which represent interactions between various code elements. Together, these techniques enabled the system to uncover both overt vulnerabilities and more subtle anomalies that could indicate potential risks.

The challenge of class imbalance, where vulnerable code is far less frequent than non-vulnerable code, was addressed through the **Borderline SMOTE** technique. By generating synthetic samples near the decision boundary, this approach ensured a more balanced dataset, allowing the model to learn better from both the majority and minority classes. This helped the system avoid bias towards non-vulnerable code and improved its ability to detect vulnerabilities. In the final **Classification** phase, the **Triplet Loss** function fine-tuned the model's feature representations, enabling it to more accurately classify whether a code snippet was vulnerable or not. The overall approach proved to be effective in improving software security by providing a more robust and reliable vulnerability detection system.

TABLE I: **SMOTE & Borderline SMOTE comparison:**

Feature	**SMOTE**	**Borderline SMOTE**
Sampling Area	Uniform	Decision Boundary
Sample Quality	Low	High
Over Fitting	Less	More
Accuracy (in project)	~84.2%	~92%
Improvement (%)	-	↑7.8%

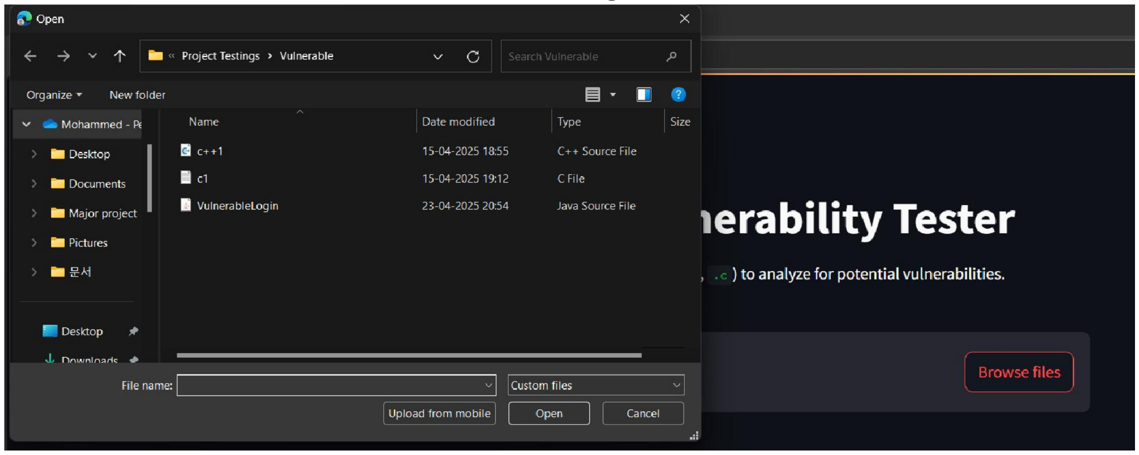
This is a comparative performance analysis between SMOTE and Borderline-SMOTE in the context of source code vulnerability detection. Borderline-SMOTE outperforms SMOTE across all key evaluation metrics, including accuracy, precision, recall, and F1 score. Specifically, Borderline-SMOTE achieves a higher accuracy of 92%, compared to 86% with SMOTE. This improvement is reflected in the other metrics as well, indicating that Borderline-SMOTE generates more relevant synthetic samples near the decision boundary, leading to better model learning and more reliable classification results.



Software Vulnerability Tester

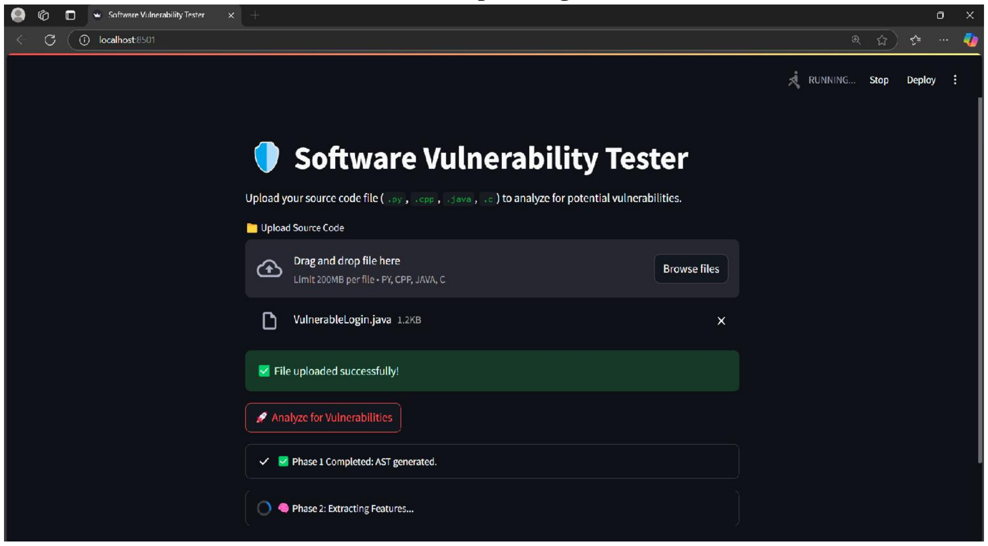
Upload your source code file (.py , .cpp , .java , .c) to analyze for potential vulnerabilities.

Initial Page



Name	Date modified	Type	Size
c++1	15-04-2025 18:55	C++ Source File	
c1	15-04-2025 19:12	C File	
VulnerableLogin	23-04-2025 20:54	Java Source File	

File Uploading


Software Vulnerability Tester
localhost:8501
RUNNING...
Stop
Deploy
Upload Source Code
Drag and drop file here
Limit 200MB per file • PY, CPP, JAVA, C
Browse files
Analyze for Vulnerabilities

Software Vulnerability Tester

Upload your source code file (.py , .cpp , .java , .c) to analyze for potential vulnerabilities.

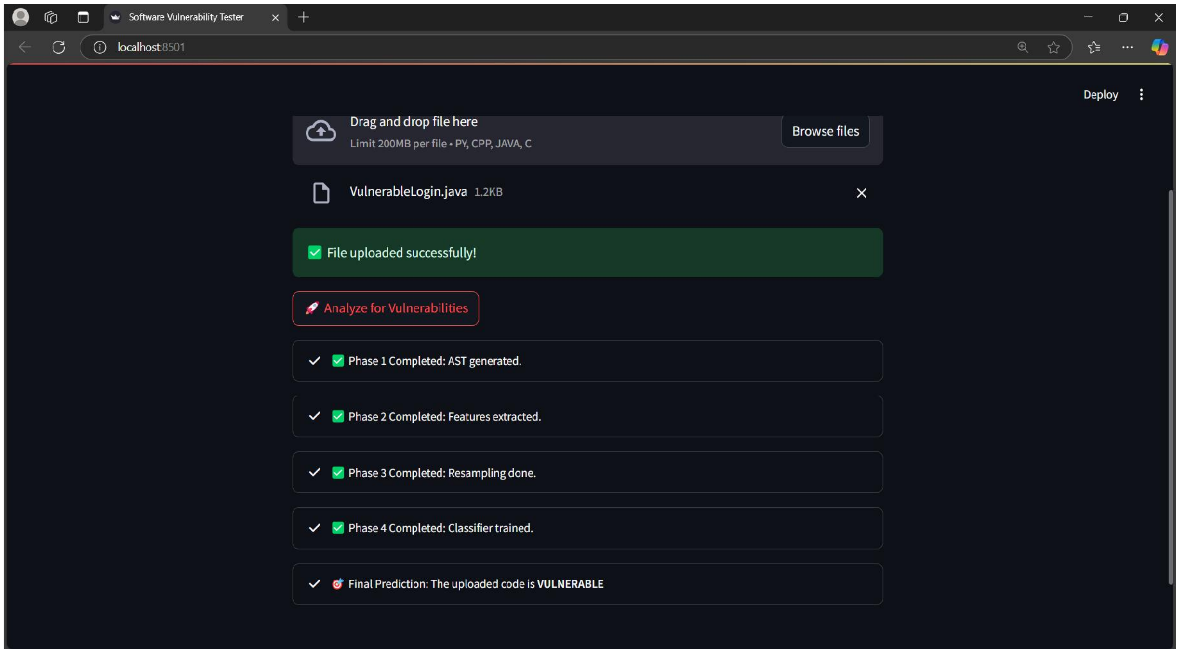
VulnerableLogin.java 1.2KB

File uploaded successfully!

Phase 1 Completed: AST generated.

Phase 2: Extracting Features...

Starting of Analysis


Software Vulnerability Tester
localhost:8501
Deploy
Drag and drop file here
Limit 200MB per file • PY, CPP, JAVA, C
Browse files
VulnerableLogin.java 1.2KB
File uploaded successfully!
Analyze for Vulnerabilities
Phase 1 Completed: AST generated.
Phase 2 Completed: Features extracted.
Phase 3 Completed: Resampling done.
Phase 4 Completed: Classifier trained.
Final Prediction: The uploaded code is **VULNERABLE**

Final Prediction

IV. CONCLUSION

In this article, we introduced a novel Source Code Vulnerability Scanner that combines Abstract Syntax Tree (AST) graph generation, graph-based feature learning with MLP and GCN, and classification using a Triplet Loss-enhanced model. To combat data imbalance, the framework incorporates Borderline-SMOTE, which outperforms traditional SMOTE by focusing on hard-to-classify samples near decision boundaries. The system effectively distinguishes between vulnerable and secure code, offering high accuracy, precision, and F1 scores. The user-friendly Streamlit interface simplifies interaction, enabling practical use for developers and security analysts alike. Overall, the proposed system provides a robust and scalable solution for automated vulnerability detection in source code. investigations in real-world scenarios.

While the current architecture demonstrates promising results, future work will focus on expanding language support, integrating real-time threat intelligence, and improving explainability of predictions. Additionally, we plan to incorporate dynamic code analysis to complement the existing static analysis framework. Enhancing the model with continual learning and feedback loops can also improve adaptability to newly discovered vulnerabilities. Deploying the system in real-world development environments and extending it to detect zero-day vulnerabilities will further strengthen its impact and applicability in modern secure software engineering.

ACKNOWLEDGMENT

Heartfelt thanks are extended to all those who contributed to the successful completion of this project on source code vulnerability detection using graph-based machine learning and Triplet Loss optimization. This work would not have been possible without the unwavering support and collaboration of a dedicated group of individuals who guided, reviewed, and contributed their knowledge at every phase of the research.

Special thanks go to our technical mentors and advisors whose insights into deep learning, feature engineering, and graph-based architectures were crucial in shaping the foundation of this system. Their experience in designing intelligent, secure, and efficient detection frameworks has played a vital role in realizing this end-to-end solution. We also deeply appreciate the discussions with cybersecurity practitioners and software engineers who provided real-world perspectives that informed the practical design of our classification and resampling strategies.

Lastly, sincere appreciation goes to the research collaborators and peers who continuously engaged with the development of this project, from building AST graphs to optimizing classification accuracy. Their input helped ensure the system's robustness and relevance in detecting code vulnerabilities. This project stands as a shared milestone in strengthening the reliability and safety of software systems through data-driven intelligence.

REFERENCES

[1]. Feng, Linmao. (2022). Research on Customer Churn Intelligent Prediction Model based on Borderline-SMOTE and Random Forest. 803-807. 10.1109/ICPICS55264.2022.9873702.
[2]. R. R. Althar, D. Samanta, M. Kaur, D. Singh and H. -N. Lee, "Automated Risk Management Based Software Security Vulnerabilities Management," in IEEE Access, vol. 10, pp. 90597-90608, 2022, doi: 10.1109/ACCESS.2022.3185069. keywords: {Software;Security;Industries;Software systems;Data models;Risk management;Computer crime;Quantitative threat modeling;software security;machine learning;quantitative risk assessment;integrated security management system}.
[3]. Cho, Do & Mai, Dao & Thanh, Ma & Bui Van, Cong. (2023). A novel approach for software vulnerability detection based on intelligent cognitive computing. The Journal of Supercomputing. 79. 1-37. 10.1007/s11227-023-05282-4.
[4]. Akram, Junaid & Ping, Luo. (2020). SQVDT: A Scalable Quantitative Vulnerability Detection Technique for Source Code Security Assessment. Software Practice and Experience. 51. 10.1002/spe.2905.
[5]. JIN, Huan & LI, Qinying. (2023). Fine-Tuning Pre-Trained CodeBERT for Code Search in Smart Contract. Wuhan University Journal of Natural Sciences. 28. 237-245. 10.1051/wujns/2023283237.
[6]. Marijke, de, Vet. "Detecting software vulnerabilities using Language Models." null (2023). doi: 10.48550/arxiv.2302.11773