Grenze International Journal of Engineering and Technology, June Issue

SekiAto: Leveraging Language Models for Software Security Vulnerability

Shruti Phutane[1], Yash Zagekar[2], Sumeet Darekar[3], Pratiksha Wanave[4] and Vijayalaxmi Kanade[5]
[1-5]Artificial Intelligence and Data Science Department, PVG's COET & GKP(W)IoM, Pune, Maharashtra, India
Email: {shrutiphutane28, yashzagekar845, sumeetdarekar2003, wanavepratiksha12, vijayalaxmi.jeure}@gmail.com

***Abstract*—Software code forms the backbone of modern applications and websites, yet ensuring its security remains a complex challenge. This research presents *SekiAto*, a novel approach to identifying and mitigating vulnerabilities in source code, particularly within PHP, JavaScript, and C/C++ environments. The proposed system builds upon the strengths of transformer-based language models by fine-tuning DistilRoBERTa and CodeBERT to enhance their effectiveness in vulnerability detection tasks. Unlike earlier models that often suffer from elevated false positive rates and struggle with identifying sophisticated vulnerabilities, *SekiAto* aims to bridge this gap. The model is trained on a carefully curated dataset that draws from publicly available repositories and verified sources of security flaws, ensuring both relevance and reliability in training data. DistilRoBERTa is specifically optimized for analyzing C/C++ code, while CodeBERT is tailored to handle vulnerabilities in PHP and JavaScript. This domain-specific fine-tuning contributes to notable improvements in detection performance, especially in terms of precision and recall. The approach highlights the effectiveness of fine-tuning transformer-based models using a domain-specific dataset, contributing to more accurate and efficient vulnerability detection. This research has strong implications for improving the security assessment process and enabling early detection of potential threats in software systems.**

***Index Terms*— Source Code Analysis, Security Vulnerability Detection, Advanced Language Processing, DistilRoBERTa Model, CodeBERT Model, Automated Code Review, Vulnerability Mitigation.**

I. INTRODUCTION

In the modern software development landscape, security vulnerabilities pose a critical threat, with the potential for severe financial and reputational damage. As reliance on software increases in both business and daily life, effective vulnerability detection becomes essential. Traditional methods like manual code reviews and static analysis tools often struggle with high false positive rates, leading to wasted resources and overlooked threats. In contrast, this research introduce *"SekiAto: A Vulnerability Sentinel"*, an AI-driven tool that leverages advanced transformer-based models to enhance vulnerability detection across PHP, C/C++, and JavaScript. While existing tools such as VulBERTa are limited to detecting vulnerabilities specifically in C/C++ code, SekiAto provides a more holistic approach by extending detection capabilities to PHP, JavaScript, and C/C++. Unlike VRepair, which centers on repairing vulnerabilities using transfer learning methods, SekiAto emphasizes proactive identification through advanced transformer-based techniques, incorporating the latest developments in artificial intelligence to enhance early detection accuracy.

Although static code analysis remains a valuable strategy for uncovering vulnerabilities in the early stages of

Grenze ID: 01.GIJET.11.2.603

development [4], [10], many conventional tools still prioritize C/C++ environments, offering limited support for other languages. Recent innovations, such as Graph Neural Networks (GNNs) paired with Program Dependence Graphs (PDGs), have shown potential in reducing false positives and improving detection precision [1], yet their application often remains narrowly focused—primarily on PHP. For instance, VulEye, which leverages Sub-Dependence Graphs (SDGs), has reported an impressive F1-score of 99% for PHP code [3], but similar performance across broader languages remains unproven. Similarly, deep learning models targeting SQL injection vulnerabilities in PHP have achieved high accuracy, yet concerns persist around their ability to generalize across different programming paradigms [2].

When it comes to JavaScript, the challenge intensifies. Its widespread adoption and potential for code obfuscation complicate the detection of malicious scripts [9]. Techniques such as Doc2Vec and deep neural networks have been applied to JavaScript bytecode analysis, showing promise, but often fall short when confronted with obfuscated or dynamically executed code. Tools like JaSt employ static analysis to detect JavaScript threats with high accuracy but may overlook runtime behaviors critical for comprehensive analysis [8].

SekiAto stands out by bridging these gaps. Through fine-tuning DistilRoBERTa for C/C++ and CodeBERT for PHP and JavaScript, it delivers a robust detection mechanism capable of identifying subtle and complex vulnerabilities that other tools may miss. By addressing the high false positive rates and limited generalizability seen in prior research, SekiAto contributes meaningfully to the field of software security. This work underscores the importance of integrating AI-powered detection models into the development lifecycle, enabling developers to take preventive action against vulnerabilities and reinforcing the foundation for more secure digital systems.

II. Related Work

This section presents a review of related studies focusing on software vulnerability detection using deep learning techniques and, more broadly, on pre-trained models for representing programming languages.

A. Neural Network approach for detecting PHP vulnerabilities

Recent efforts in vulnerability detection for PHP code have increasingly leveraged deep learning, particularly transfer learning. In this approach, models pre-trained on large-scale general-purpose code datasets are fine-tuned with PHP-specific data, improving detection accuracy while lowering the computational cost of training from scratch [4]. Simultaneously, researchers have examined vulnerabilities linked to PHP's object-oriented features, which introduce complex interdependencies among classes and objects. Conventional methods often struggle to identify flaws rooted in these structural elements. Studies addressing this challenge emphasize the need to model object hierarchies and interactions, showing that deeper insights into inheritance and encapsulation lead to more effective vulnerability detection [14].

In extending structural analysis, Graph Neural Networks (GNNs) have been adopted to capture both the syntactic and semantic aspects of PHP code. By modeling code as graphs that reflect its logical and structural organization, GNN-based approaches can detect vulnerabilities missed by purely syntax-driven techniques. This method improves the ability to recognize contextual dependencies and trace data flows—key factors in exposing complex security flaws [3]. Supporting these innovations is the development of specialized datasets. A newly released benchmark dataset compiles a broad range of PHP vulnerabilities, serving as a crucial resource for training and evaluating detection models. This dataset plays a significant role in promoting the creation of more resilient and generalizable security tools for PHP applications [5].

B. Fine-Tuned models for detecting JavaScript Vulnerabilities

JavaScript's dynamic nature and widespread use in client-side development introduce significant challenges for vulnerability detection. Deep learning approaches, particularly those analyzing bytecode, have shown effectiveness in identifying malicious patterns. Sequence-based neural networks, capable of learning recurring features, are especially suited to detect evolving forms of obfuscated attacks [6].

Obfuscation remains a key tactic for evading traditional defenses. To counter this, recent studies have focused on training models to detect concealed or transformed scripts, emphasizing the need for robust systems that remain accurate despite adversarial evasion strategies [7]. Syntactic analysis has also proven effective, with models leveraging JavaScript's grammar and heuristic rules to detect both overt and subtle threats [8].

A notable development in this area is the use of Abstract Syntax Tree (AST) features within machine learning workflows. AST-based representations offer deeper insight into code structure and logic, enabling models to identify vulnerabilities that simpler techniques may overlook. This method enhances contextual understanding

and improves detection accuracy, particularly in light of JavaScript's flexible execution environment [9]. Together, these strategies—bytecode analysis, obfuscation detection, syntactic parsing, and AST integration—form a comprehensive framework for addressing the complexity of JavaScript vulnerabilities, leading to more resilient and precise security solutions.

C. C++ Vulnerability Detection with Large language model

C++ is a core language for system-level development, but its complexity—ranging from pointer manipulation to manual memory management—makes vulnerability detection particularly challenging. Comparative analyses of static analysis tools reveal that effective detection in C++ requires tailored solutions and highlight areas for framework refinement [12].

In contrast to rule-based approaches, deep learning has emerged as a powerful method for identifying vulnerabilities in C and C++ code. Neural models trained on relevant datasets can detect nuanced structural and semantic patterns that traditional techniques often miss, resulting in improved accuracy [13].

Hybrid architectures, such as quantum convolutional neural networks originally designed for Java, have recently been explored for cross-language vulnerability detection. Their integration of diverse feature extraction methods makes them adaptable to C++, offering a promising path toward more generalizable tools [11]. Ongoing evaluations of C/C++ vulnerability detectors continue to expose performance gaps and detection blind spots. These findings stress the importance of building tools that can accommodate the increasing complexity and scale of modern C++ systems [10].

A comparative overview of existing methodologies, their performance parameters, and efficiency limitations is presented in Table I, highlighting key challenges such as scalability issues, inefficient resource utilization, and the underperformance of models on complex or obfuscated code.

III. SEKIATO METHODOLOGY

This section outlines the detailed methodology adopted in the proposed system for detecting vulnerabilities in JavaScript, PHP, and C/C++ code snippets using transformer-based models such as CodeBERT and DistilRoBERTa.

A. Data Collection

For this study, the data is collected from code snippets from multiple reliable sources. These included open-source repositories like GitHub, public vulnerability databases, and additional samples obtained through targeted web crawling. The dataset spans three widely-used programming languages — JavaScript, PHP, and C/C++, chosen due to their popularity in both web and system-level development, as well as their frequent exposure to security threats.

Each snippet in the dataset is labeled as either:

- *Vulnerable (1):* Code that contains one or more common security flaws, such as cross-site scripting (XSS), command injection, SQL injection, or buffer overflows.

Non-vulnerable (0): Code that is considered secure, with no known exploitable issues.

TABLE I. METHODOLOGY COMPARISON WITH PAST RESEARCH

Methodology	Performance Parameters	Efficiency Issues
Hybrid Graph Neural Network (GNN)	Combines traditional program analysis with machine learning, capturing syntactic and semantic features of code. Lower false positives with PDG, but scalability issues for large codebases.	Lack of Parallelization, Pruning, and Suboptimal Data Structures
Deep Learning with Natural Language Processing (NLP)	Effective in detecting vulnerabilities by understanding code context through an intermediate representation, but training requires significant resources and can cause latency	Large and complex machine learning models often face challenges in deployment due to their excessive size and slow inference times. Without applying optimization techniques, these models become resource-intensive and inefficient, particularly in production environments..
Machine Learning with Abstract Syntax Tree (AST) and Doc2Vec	Fast feature extraction and classification with clear AST representations, but less effective against obfuscated or minified code.	Inaccurate Detection of Complex Code Vulnerabilities Due to Insufficient Preprocessing and Underutilization of Advanced Model

Methodology	Performance Parameters	Efficiency Issues
Graph Neural Network (GNN) with Program Dependence Graph (PDG)	High accuracy using SubDependence Graphs (SDG), but computational intensity can hinder scalability for large projects.	Slow and Expensive Large-Scale Graph Analysis Due to Inefficient Resource Utilization and Lack of Optimization

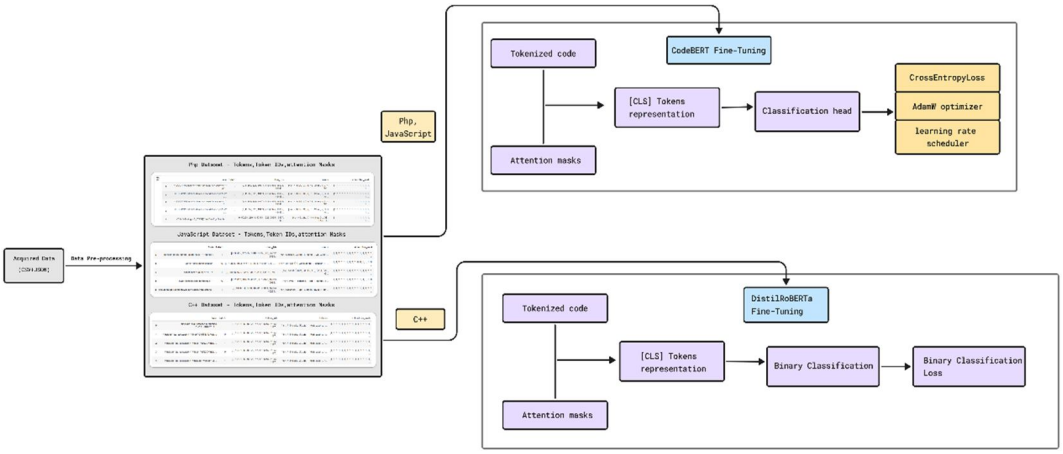


Figure 1. Generation of Token IDs and Attention masks and Fine tuning with custom dataset

The overall fine-tuning pipeline for CodeBERT and DistilRoBERTa models, along with preprocessing steps applied to PHP, JavaScript, and C/C++ datasets, is depicted in Fig. 1. This workflow demonstrates how input source code is tokenized, converted into token IDs and attention masks, and subsequently passed through the transformer models.

B. Data Preprocessing and Feature Representation

- *Tokenization:-* Tokenization is carried out using the respective pre-trained model's tokenizer, with CodeBERT Tokenizer applied to JavaScript and PHP code snippets, and DistilRoBERTa Tokenizer used for C/C++ as well as a lightweight alternative across all languages for comparative analysis. Tokenization involves splitting the code into subword tokens, including identifiers, operators, and language-specific syntax (e.g., echo, eval, system, document.write, malloc).
- *Attention Mask Generation:-* An attention mask is created for each tokenized sequence. The mask is a binary array where a value of 1 represents a real token that should be attended to by the model, while a value of 0 indicates padding added to maintain a fixed sequence length. This guides the transformer model in focusing only on the meaningful parts of the input while ignoring the padded tokens during self-attention operations.
- *Padding and Truncation:-* All sequences are padded or truncated to a fixed length (512 tokens), ensuring consistent input dimensions. This allows for efficient batching and processing during model training and inference. Padding ensures shorter sequences match the input size requirement, while truncation prevents excessively long inputs from exceeding memory limits.

C. Custom Dataset Construction

To manage the preprocessed data effectively, a custom dataset class is constructed using PyTorch. This dataset object returns, for each item:

- *input_ids:* Tokenized and encoded numerical representation of the code.
- *attention_mask:* Corresponding attention mask for the input.
- *label:* Ground truth class label (0 for non-vulnerable, 1 for vulnerable).

This design aligns with PyTorch's DataLoader, enabling efficient batch processing, GPU acceleration, and shuffling. The modular dataset abstraction enhances scalability and supports experimentation with different models and batch sizes.

D. Data Splitting

Data splitting was conducted strategically to balance model learning and evaluation integrity. Depending on the complexity of the model and size of the dataset, splits such as 70:30, 75:25, or 80:20 were applied. The training portion was used to fine-tune the model, while the test set was reserved for final performance evaluation. Stratified sampling was applied to maintain class balance across both subsets.

E. Fine-tuning

a. *CodeBERT Fine-Tuning*

CodeBERT is a transformer-based model pre-trained on a corpus of source code and natural language across multiple programming languages. It is particularly effective for code-related tasks due to its training on a wide variety of code syntax and semantics.

For this work, CodeBERT is fine-tuned as follows:

- The tokenized code is passed to the model along with attention masks.
- The representation of the special [CLS] token is extracted, which captures the contextual information of the entire sequence.
- This representation is fed into a fully connected classification head, which outputs logits corresponding to the two classes (vulnerable or non-vulnerable).
- The model is trained using CrossEntropyLoss, and parameters are optimized using the AdamW optimizer with a learning rate scheduler for dynamic adjustment.

b. DistilRoBERTa Fine-Tuning

DistilRoBERTa is a compact and efficient variant of the RoBERTa model that retains approximately 95% of its performance while being 40% smaller and offering 60% faster inference speed. Although not pre-trained specifically on source code, DistilRoBERTa demonstrates strong generalization capabilities when fine-tuned on a labeled dataset of vulnerable and non-vulnerable code samples.

It follows a similar fine-tuning pipeline:

- Input code snippets are preprocessed and tokenized using the DistilRobertaTokenizerFast from Hugging Face Transformers. Each sample is transformed into input_ids, attention_mask, and associated binary labels (0 for safe, 1 for vulnerable).
- A custom PyTorch Dataset class is used to load tokenized data and labels efficiently. Padding and truncation are handled during tokenization to ensure uniform input length. DistilBertForSequenceClassification with a binary classification head is used. The [CLS] token embedding is used for final prediction.
- The model is trained using the AdamW optimizer and binary cross-entropy loss.

Fig. 2 illustrates the end-to-end workflow adopted for vulnerability detection using fine-tuned transformer models. Initially, code samples are gathered and curated into a structured dataset, which undergoes preprocessing, the processed data is fed into transformer-based models, CodeBERT for PHP and JavaScript, and DistilRoBERTa for C/C++, which are fine-tuned to classify code snippets as vulnerable or non-vulnerable. The system culminates in the generation of structured reports highlighting the detected language, vulnerabilities and its severity.

IV. Model Training Algorithm

The *checkVulnerability* function is designed to detect security vulnerabilities in code snippets written in multiple programming languages, including PHP, JavaScript, C, and C++. It utilizes language-specific pre-trained large language models, namely CodeBERT and DistilRoBERTa.

The function follows these key steps:

Language Detection:- The process begins with detecting the programming language of the input code using the *detect_language()* method. Only code written in PHP, JavaScript, C, or C++ is supported. If the language is not supported, the function returns an “unsupported language” message.

Syntax Validation:- The function begins by validating the syntax of the input PHP code using the *code_syntax_is_valid()* method. This step ensures that only syntactically correct code proceeds for analysis, minimizing false positives from invalid code.

Tokenization and Model Selection:- Based on the detected language, the appropriate tokenizer and model are selected:

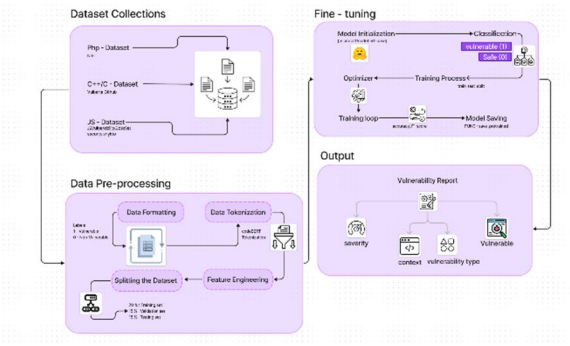


Figure 2. Architecture of SekiAto: A Vulnerability Sentinel

```
function checkVulnerability(code)
    language = detect_language(code)
    if language ∉ {PHP, JavaScript, C/C++} then
        return 'unsupported language'
    end if

    if code_syntax_is_valid(code, language) then
        if language ∈ {PHP, JavaScript} then
            tokenized_code = CodeBERT_tokenize(code)
            prediction = CodeBERT_model.predict(tokenized_code)
        else if language ∈ {C, C++} then
            tokenized_code = DistilRoBERTa_tokenize(code)
            prediction = DistilRoBERTa_model.predict(tokenized_code)
        end if

        if prediction == vulnerable then
            generate_pdf_report(code, prediction)
            return 'vulnerable'
        else
            return 'safe'
        end if
    else
        return 'syntax error'
    end if
end function
```

Figure 3. SekiAto model training algorithm

For PHP and JavaScript, the code is tokenized using *CodeBERT_tokenize()* and analyzed using the *CodeBERT_model*.
For C and C++, the code is tokenized using *DistilRoBERTa_tokenize()* and analyzed using the *DistilRoBERTa_model.*
Tokenization converts raw code into a structured format suitable for processing by the respective language model.
Vulnerability Prediction:- The selected model predicts whether the input code is *"vulnerable"* or *"safe"* based on patterns learned from a labeled dataset containing vulnerable and non-vulnerable code snippets.
Vulnerability Handling:-
If the prediction indicates that the code is *"vulnerable"*, the function generates a detailed PDF report using the *generate_pdf_report()* method. This report includes the input code and the vulnerability assessment, providing actionable insights to the user.
If the code is deemed *"safe"*, the function directly returns this classification.
1. *Error Handling.* If the code does not pass syntax validation, the function returns a *"syntax error"* message to inform the user about the issue before proceeding further.
As shown in Fig. 3, the pseudocode for SekiAto's inference algorithm demonstrates a structured decision-making process based on language detection and vulnerability prediction. The input is first checked for compatibility with supported languages, and then validated for syntax correctness. Based on the language, appropriate tokenizers and transformer models are applied. The corresponding model then predicts whether the input is vulnerable. If a vulnerability is detected, a detailed PDF report is generated; otherwise, the code is marked as safe.

V. Model Performance

To assess the effectiveness of transformer-based models in detecting code vulnerabilities, three fine-tuned models were evaluated across JavaScript, PHP, and C/C++ datasets. The models were trained on a labeled dataset comprising both vulnerable and non-vulnerable code snippets. Performance was measured using standard classification metrics on a held-out test set, and the results are summarized in Table II.
In vulnerability detection for JavaScript, the CodeBERT model delivered stable classification results with balanced accuracy across both classes. However, the model exhibited a moderate false positive rate of 7.50%, indicating a tendency to incorrectly flag secure code as vulnerable in a notable number of cases. Despite maintaining high accuracy, such a false positive rate may hinder its effectiveness in scenarios where precision is critical.
For PHP, CodeBERT displayed a conservative detection approach. While it achieved zero false positives, this came at the expense of failing to identify any vulnerable code samples. The model was highly biased towards the non-vulnerable class, leading to skewed classification performance and severely reduced overall reliability. The absence of false positives is outweighed by its complete lack of sensitivity to actual vulnerabilities.
DistilRoBERTa on C/C++ code provided a more refined balance between sensitivity and precision. It achieved high classification accuracy while maintaining an exceptionally low false positive count. With only two incorrect classifications among a large set of non-vulnerable instances, the model demonstrated strong practical viability, particularly in minimizing unnecessary alerts while still detecting actual threats effectively

TABLE II. MODEL-WISE PERFORMANCE COMPARISON FOR VULNERABILITY DETECTION IN JS, PHP, AND C/C++

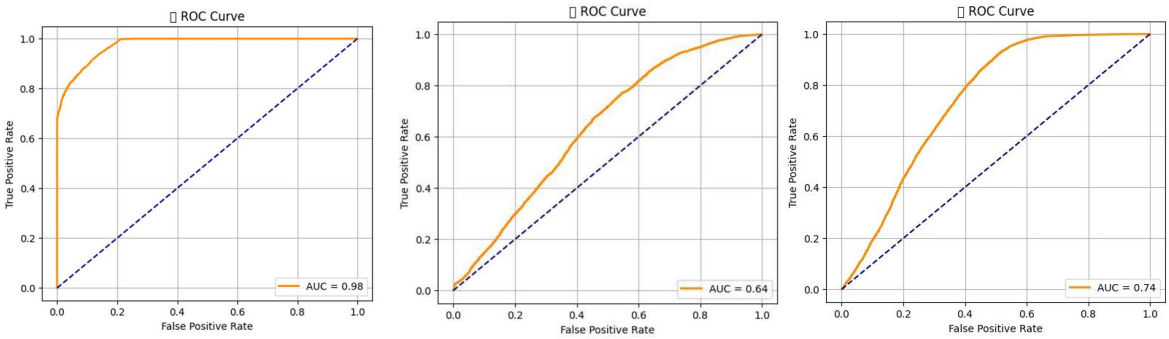
Model	Class	Precision	Recall	F1-Score	Support
CodeBERT (JavaScript)	Non Vulnerable JS	0.89	0.92	0.91	6476
	Vulnerable JS	0.91	0.86	0.88	5565
	Accuracy			0.90	12,041
	Macro Avg	0.90	0.89	0.90	12,041
	Weighted Avg	0.90	0.90	0.90	12,041
CodeBERT (PHP)	Non Vulnerable PHP	0.70	1.00	0.82	10294
	Vulnerable PHP	0.00	0.00	0.00	4478
	Accuracy			0.7	14772
	Macro Avg	0.35	0.5	0.41	14772
	Weighted Avg	0.49	0.7	0.57	14772
DistilRoBERTa (C/C++)	Non Vulnerable C/C++	0.94	1.00	0.97	20044
	Vulnerable C/C++	1.00	0.78	0.88	6301
	Accuracy			0.95	26,345
	Macro Avg	0.97	0.89	0.92	26,345
	Weighted Avg	0.95	0.95	0.95	26,345

Fig. 4 provides a visual comparison of the models 'performance through ROC (Receiver Operating Characteristic) curves, along with corresponding Area Under the Curve (AUC) scores. The CodeBERT model for JavaScript exhibited superior performance, achieving an AUC of 0.98, which confirms excellent discrimination capability between vulnerable and non-vulnerable code. The curve closely aligns with the top-left corner of the ROC space, indicating minimal false positive and false negative rates.

In the case of PHP, CodeBERT registered a significantly lower AUC of 0.64, consistent with the observed classification metrics. The curve remains close to the diagonal line, suggesting that the model's ability to differentiate between classes is only marginally better than random guessing. For C/C++, the DistilRoBERTa model achieved an AUC of 0.74, indicating better discrimination than the PHP model but still not reaching the JavaScript model's level. The curve suggests a relatively balanced trade-off between true positives and false positives, aligning well with the high recall and precision observed in the classification report.

VI. CONCLUSION

This study successfully applied CodeBERT and DistilRoBERTa for automated vulnerability detection in PHP, JavaScript, and C/C++ codebases. CodeBERT, with its code-specific training, excels in PHP and JavaScript, while DistilRoBERTa provides a lightweight yet capable solution for C and C++ analysis. The approach exhibits strong performance by learning structural and semantic code features well, as well as adjusting to each language's special security problems. This system has the potential to significantly reduce security risks in production environments and continued optimization and expansion will further align this work with industry demands. Even with these improvements, there are still some limitations—such as high computational expense, difficulty in generalizing to unseen code patterns, and the limited interpretability of deep learning models. These constraints provide various directions for further research: achieving scalability with more extensive datasets, real-time detection, and the integration of explainability methods. Collectively, these efforts contribute to more robust detection methodologies, paving the way for safer and more secure software applications across diverse programming languages.



(a) (b) (c)

Figure 3. ROC-curve of Vulnerability Detection Models — (a) CodeBERT (JavaScript), (b) CodeBERT (PHP), (c) DistilRoBERTa (C/C++)

REFERENCES

[1] S. Zhu, H. Shi, Z. Zhou, Z. Xu, and X. Cheng, "A hybrid graph neural network approach for detecting PHP vulnerabilities," Proceedings of the 2022 ACM on Asia Conference on Computer and Communications Security (AsiaCCS), pp. 34-45, 2022.

[2] J. Liu, Y. Zheng, and X. Wu, "Towards a deep learning model for vulnerability detection on web application variants," IEEE International Conference on Data Mining (ICDM), pp. 678-687, 2020.

[3] M. Gao, Q. Zhang, and J. Liu, "VulEye: A novel graph neural network vulnerability detection approach for PHP applications," Journal of Information Security and Applications, vol. 58, 2022.

[4] P. Bertolino, M. Orru, and A. Di Pietro, "VulBERTa: Simplified source code pre-training for vulnerability detection," Proceedings of the 2021 IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 99-108, 2021.

[5] W. Xu, Y. Li, and D. Wang, "DiverseVul: A new vulnerable source code dataset for deep learning-based vulnerability detection," Proceedings of the 2022 IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 58-67, 2022.

[6] M. Dey and A. Singh, "Deep neural networks for malicious JavaScript detection using bytecode sequences," IEEE Transactions on Information Forensics and Security, vol. 16, pp. 135-147, 2021.

[7] R. Perez and P. Garcia, "Detection of obfuscated malicious JavaScript code," Journal of Computer Virology and Hacking Techniques, vol. 17, pp. 45-56, 2021.

[8] L. Wang, Z. Lin, and J. He, "JaSt: Fully syntactic detection of malicious (obfuscated) JavaScript," Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security (CCS), pp. 123-134, 2022.

[9] A. Kumar, S. Banerjee, and T. Arora, "A machine learning approach to detection of JavaScript-based attacks using AST features and paragraph vectors," ACM Transactions on Privacy and Security, vol. 25, no. 3, pp. 43-56, 2022.

[10] A. R. Gupta and N. Bhattacharyya, "Neural transfer learning for repairing security vulnerabilities in C code," International Journal of Advanced Computer Science and Applications (IJACSA), vol. 13, no. 6, pp. 99-106, 2022.

[11] H. Liu, M. Zhang, and C. Sun, "Vulnerability detection in Java source code using a quantum convolutional neural network with self-attentive pooling," Journal of Systems and Software, vol. 179, pp. 104-112, 2021.

[12] D. Patel, M. Kumar, and S. Ray, "A comparative study of static code analysis tools for vulnerability detection in C/C++ and Java source code," ACM Transactions on Software Engineering and Methodology, vol. 30, no. 2, pp. 56-69, 2021.

[13] Huang, Zhen, and Amy Aumpansub. "Vulnerability Detection in C/C++ Code with Deep Learning." arXiv preprint arXiv:2405.12384 (2024).

[14] Nashaat, Mona, Karim Ali, and James Miller. "Detecting security vulnerabilities in object-oriented php programs." 2017 IEEE 17th International Working Conference on Source Code Analysis and Manipulation (SCAM). IEEE, 2017.