CodeBERT-Based Vulnerability Detection

Xinyi Liu

December 2024

1 Introduction

Software vulnerabilities pose a significant threat to the security and reliability of digital systems, often resulting in severe consequences such as unauthorized access and data breaches. Notable incidents like the 2017 Equifax breach, which exposed sensitive information of 143 million individuals, and the 2023 MOVEit breach, which exploited a flaw in data transfer software, highlight the urgent need to detect and mitigate vulnerabilities early in the software development lifecycle [1, 2]. As modern software grows in complexity and scale, traditional methods struggle to keep pace, leaving systems increasingly exposed.

Current vulnerability detection methods, including manual code reviews and static analysis tools, have inherent limitations. Manual reviews are time-intensive, error-prone, and inefficient for large codebases. Static analysis tools, while automated, depend on predefined rules and patterns, making them less effective against novel or evolving vulnerabilities. These limitations contribute to high false positives and false negatives rates, reducing their overall reliability [3, 4]. The growing sophistication of software systems demands more adaptive and accurate detection mechanisms.

To address these challenges, this project employs CodeBERT [5], a transformer-based language model pre-trained on large datasets of source code and natural language. By fine-tuning CodeBERT for binary classification, we classify code snippets as safe or vulnerable. Unlike traditional tools, CodeBERT captures structural and contextual nuances in code, enabling it to detect vulnerabilities across diverse programming languages and patterns.

This approach combines the strengths of advanced language modeling with practical application in secure software development. Enhancing adaptability and improving scalability, our solution offers an innovative framework for strengthening software security in an era of increasingly complex digital systems.

2 Related Works

Code vulnerability analysis is essential for identifying and mitigating security flaws in software systems. Traditional methods, such as static and dynamic anal-

ysis, rely on predefined rules and patterns, making them inflexible and ill-suited for addressing evolving vulnerabilities. Machine learning (ML) approaches offer greater adaptability by learning patterns in code, enabling enhanced detection capabilities. However, current ML-based solutions primarily focus on limited programming languages, restricting their broader applicability.

2.1 ML Approaches by Language

C/C++ poses unique challenges in vulnerability detection due to its low-level memory management, which often leads to issues such as buffer overflows and null pointer dereferences. Techniques like Abstract Syntax Trees (ASTs) and Control Flow Graphs (CFGs) are widely used to represent syntactic and control-flow patterns in code to address these challenges. Zhou et al. demonstrated that AST-based models can significantly improve the detection of structural vulnerabilities in C/C++ code [6]. Similarly, Ahmed et al. utilized Sequential Graph Neural Networks (SEGNNs) to jointly process ASTs and CFGs, enhancing the detection of complex memory-related vulnerabilities [7].

Python, characterized by dynamic typing and extensive library support, introduces challenges in detecting vulnerabilities related to data misuse and injection attacks. Data Flow Graphs (DFGs) have proven effective in capturing data dependencies across variables and functions, enabling the identification of improper data handling. Steenhoek et al. successfully used DFGs to detect vulnerabilities involving sensitive data exposure [8].

JavaScript, commonly used in web applications, presents unique difficulties due to its asynchronous and event-driven nature. Code Property Graphs (CPGs), which integrate ASTs, CFGs, and DFGs, provide a comprehensive view of JavaScript code, aiding in detecting vulnerabilities such as cross-site scripting (XSS). Ferenc et al. leveraged CPGs to identify complex structural and functional dependencies, improving the detection of web-specific vulnerabilities [9].

2.2 Language Models for Vulnerability Detection

Recent advancements in ML for vulnerability detection leverage pre-trained language models tailored to analyzing source code. RoBERTa, as fine-tuned by Do et al., has demonstrated strong performance in detecting vulnerabilities in C/C++ by understanding complex code semantics [10]. Zhou et al. introduced a Large Language Model for Vulnerability Detection, which leverages extensive pretraining on security-specific datasets to generalize across languages such as Python and JavaScript [11]. CodeBERT, designed specifically for source and natural language, effectively captures syntax and semantics [5].

While these models achieve state-of-the-art performance, their focus on specific languages, such as C/C++, Python, and JavaScript, limits their applicability to a broader range of programming environments. For instance, RoBERTa excels in detecting memory-related vulnerabilities in C/C++ through hierarchical data representations, while Zhou's model performs well in identifying Python-specific vulnerabilities like data misuse. CodeBERT bridges some of

these gaps by adapting to diverse programming environments, but challenges remain in scaling its capabilities to underrepresented languages and vulnerability patterns.

3 Innovative Approach

This project adopts a CodeBERT-based machine learning model fine-tuned for binary source code classification as "safe" or "vulnerable." The approach addresses the limitations of traditional vulnerability detection techniques and aims to achieve high accuracy, precision, recall, and F1 scores through iterative optimization and comparative analysis.

3.1 Dataset Preparation

The datasets used were **CVEfixes (Concise Version)** and **DiverseVul**, carefully selected for their diverse representation of vulnerabilities across programming languages.

- **CVEfixes [12]**: A structured dataset derived from the National Vulnerability Database (NVD), focusing on file-level information. It consists of three key columns: `code`, `language`, and `safety`, allowing precise vulnerability identification.
- **DiverseVul**: A larger and more diverse dataset containing 18,945 vulnerable functions and 330,492 nonvulnerable functions across 155 CWEs and various programming languages. This dataset complements CVEfixes by providing a broader scope of vulnerability patterns.

The datasets were tokenized using the **CodeBERT tokenizer**. Tokenization was performed in batches to optimize memory usage (shown in Fig. 1), converting the source code into input IDs and attention masks for compatibility with the CodeBERT architecture.

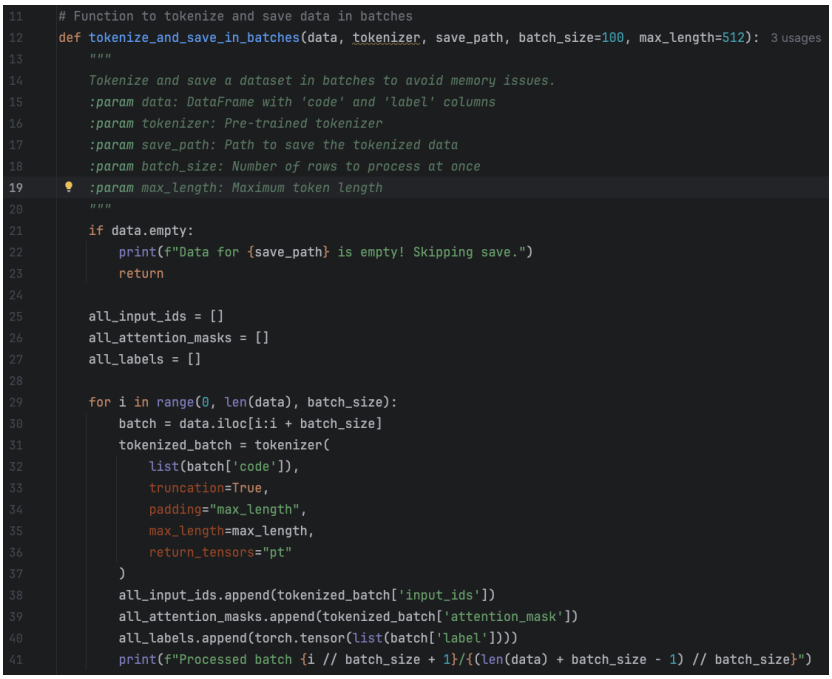
3.2 Model Fine-Tuning

The implementation leveraged **CodeBERT**, a pre-trained transformer model designed for source code and natural language processing. The fine-tuning process was iterative and included multiple adjustments to optimize model performance.

3.2.1 Initial Setup

The initial training parameters were as follows (shown in Fig. 2a):

- **Learning Rate**: 2×10^{-5}
- **Batch Size**: 8



```python
# Function to tokenize and save data in batches
def tokenize_and_save_in_batches(data, tokenizer, save_path, batch_size=100, max_length=512):  3 usages
    """
    Tokenize and save a dataset in batches to avoid memory issues.
    :param data: DataFrame with 'code' and 'label' columns
    :param tokenizer: Pre-trained tokenizer
    :param save_path: Path to save the tokenized data
    :param batch_size: Number of rows to process at once
    :param max_length: Maximum token length
    """
    if data.empty:
        print(f"Data for {save_path} is empty! Skipping save.")
        return

    all_input_ids = []
    all_attention_masks = []
    all_labels = []

    for i in range(0, len(data), batch_size):
        batch = data.iloc[i:i + batch_size]
        tokenized_batch = tokenizer(
            list(batch['code']),
            truncation=True,
            padding="max_length",
            max_length=max_length,
            return_tensors="pt"
        )
        all_input_ids.append(tokenized_batch['input_ids'])
        all_attention_masks.append(tokenized_batch['attention_mask'])
        all_labels.append(torch.tensor(list(batch['label'])))
        print(f"Processed batch {i // batch_size + 1}/{(len(data) + batch_size - 1) // batch_size}")
```

Figure 1: Implementation Code for Tokenization

- **Epochs**: 3
- **Evaluation Strategy**: Per epoch

The model was fine-tuned on the combined datasets, and the trained model was saved. However, the initial results were unsatisfactory, with precision, recall, and F1 scores all at **0** (shown in Table 1). This indicated that the model struggled to learn meaningful patterns from the datasets.

Table 1: Performance Comparison of Tuned-CodeBERT and CodeBERT Models

Model	Accuracy	Precision	Recall	F1 Score
CodeBERT	52.8%	0%	0%	0%
Tuned-CodeBERT Initial	48.8%	0%	0%	0%
Tuned-CodeBERT Updated	52.7%	47.3%	71.2%	56.8%

3.2.2 Adjustments

To address the poor initial results, the following changes were made (shown in Fig. 2b):

- The learning rate was reduced to 1×10^{-5}.

```python
training_args = TrainingArguments(
    output_dir="./results",
    evaluation_strategy="epoch",
    save_strategy="epoch",
    learning_rate=2e-5,
    per_device_train_batch_size=8,
    per_device_eval_batch_size=8,
    num_train_epochs=3,
    weight_decay=0.01,
    logging_dir="./logs",
    logging_steps=10,
    save_total_limit=2,
    load_best_model_at_end=True,
    metric_for_best_model="f1",
    greater_is_better=True,
)
```

(a) Initial Parameters Setup

```python
training_args = TrainingArguments(
    output_dir="./results",
    evaluation_strategy="epoch",
    save_strategy="epoch",
    learning_rate=1e-5,
    per_device_train_batch_size=8,
    per_device_eval_batch_size=8,
    num_train_epochs=5,
    weight_decay=0.01,
    logging_dir="./logs",
    logging_steps=10,
    save_total_limit=2,
    load_best_model_at_end=True,
    metric_for_best_model="f1",
    greater_is_better=True,
)
```

(b) Updated Parameters Setup

Figure 2: Comparison of Initial and Updated Parameters Setup

- The number of training epochs was increased to 5.

These adjustments were applied to the previously saved model, leveraging its partially trained parameters rather than starting from scratch.

3.2.3 Improved Results

After applying the adjustments, the updated model achieved significantly better results, with noticeable improvements across precision, recall, and F1-score (shown in Table 1). This highlighted the importance of hyperparameter tuning and iterative optimization in fine-tuning transformer models for code-based tasks.

3.3 Comparative Analysis with CodeBERT Baseline

To evaluate the effectiveness of the optimized model, its performance was compared with the baseline CodeBERT model:

- The **baseline CodeBERT** model yielded results similar to the first iteration of training, with poor performance across precision, recall, and F1-score (shown in Table 1).
- In contrast, the **optimized model** (learning rate: 1×10^{-5}, epochs: 5) demonstrated superior performance, validating the improvements achieved through hyperparameter tuning and dataset-specific adjustments.

3.4 Implementation Details

The training process was implemented using PyTorch and HuggingFace's Transformers library. The key steps included:

- **Data Loading**: Tokenized datasets were loaded into PyTorch `DataLoader` objects for batch processing.
- **Training Loop**: The model processed input IDs and attention masks, generating logits for binary classification. The loss function (cross-entropy) was computed, and gradients were backpropagated to update model weights.
- **Validation**: After each epoch, validation was performed to monitor performance, ensuring the model's generalization capability.

3.5 Evaluation Metrics

The model's performance was evaluated using precision, recall, F1-score, and accuracy. These metrics provided a comprehensive assessment of its ability to correctly classify code snippets while minimizing false positives and negatives. The optimized model achieved better balance across these metrics, emphasizing its ability to detect diverse vulnerabilities effectively.

4 Future Extensions

The current implementation demonstrates the potential of a fine-tuned CodeBERT model for vulnerability detection. However, there are several promising avenues for future research and development. These extensions address existing limitations while exploring practical applications to enhance the model's impact in real-world scenarios.

4.1 Model Enhancement and Generalization

While the current datasets already cover multiple programming languages, such as Python, Java, C, and PHP, future work could incorporate advanced training techniques like active learning or self-supervised learning. These approaches can potentially reduce the dependence on labeled datasets by utilizing large-scale unlabeled code repositories, enabling the model to identify vulnerabilities with minimal manual intervention.

4.2 Deployment Considerations

Deploying the model in real-world environments introduces unique challenges and opportunities. A key direction is integrating the trained model into continuous integration/continuous deployment (CI/CD) pipelines for real-time vulnerability detection during software development. This requires optimizing the

model for inference speed and ensuring compatibility with widely used platforms like Jenkins, GitHub Actions, or GitLab CI.

Moreover, improving the explainability of the model's predictions is essential. Developers need clear, actionable insights to understand and address detected vulnerabilities. Future research could leverage explainable AI (XAI) techniques to provide detailed reasoning for the model's classifications, facilitating the debugging and mitigation processes.

4.3 Interactive Feedback Systems

Future developments can integrate the model with interactive development environments (IDEs) such as Visual Studio Code or JetBrains IntelliJ to enhance usability. These integrations could provide real-time feedback by highlighting potential vulnerabilities as developers write code. Moreover, incorporating specific CVE/CWE identifiers into the model's predictions would provide more context and precision, helping developers understand the nature of the detected vulnerabilities.

4.4 Exploration of Hybrid Models

Future research can explore hybrid approaches that combine rule-based static analysis tools with machine learning-based methods. A hybrid model would leverage the strengths of both techniques—static analysis for precise rule-based detections and machine learning for adaptive, context-aware vulnerability identification. This integration has the potential to significantly reduce false positives and negatives while enhancing the model's adaptability to novel vulnerabilities.

4.5 Performance Optimization

Optimizing the model for scalability and resource efficiency is critical for real-world deployment. Techniques such as model distillation, quantization, and pruning can reduce the computational overhead during inference, enabling deployment in resource-constrained environments. Additionally, distributed training methods could be employed to handle larger datasets effectively, improving the model's scalability and responsiveness.

References

[1] J. McDonald, "Equifax Data Breach: What Went Wrong," *MIT Sloan School of Management*, 2020. [Accessed: Dec. 5, 2024].

[2] D. Goodin, "Mass exploitation of critical moveit flaw is ransacking orgs big and small," *arstechnica*, 2023.

[3] M. Alqaradaghi and T. Kozsik, "Comprehensive evaluation of static analysis tools for their performance in finding vulnerabilities in java code," *IEEE Access*, vol. 12, pp. 55824–55842, 2024.

[4] A. S. Ami, K. Moran, D. Poshyvanyk, and A. Nadkarni, “”false negative - that one is going to kill you”: Understanding industry perspectives of static analysis based security testing,” in *2024 IEEE Symposium on Security and Privacy (SP)*, pp. 3979–3997, 2024.

[5] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, e. . C. T. Zhou, Ming”, Y. He, and Y. Liu, “CodeBERT: A pre-trained model for programming and natural languages,” in *Findings of the Association for Computational Linguistics: EMNLP 2020*, (Online), pp. 1536–1547, Association for Computational Linguistics, Nov. 2020.

[6] Y. Zhou, S. Liu, J. K. Siow, X. Du, and Y. Liu, “Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks,” in *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, 2019, Vancouver, BC, Canada*, pp. 10197–10207, 2019.

[7] A. Ahmed, A. Said, M. Shabbir, and X. D. Koutsoukos, “Sequential graph neural networks for source code vulnerability identification,” *CoRR*, vol. abs/2306.05375, 2023.

[8] B. Steenhoek, H. Gao, and W. Le, “Dataflow analysis-inspired deep learning for efficient vulnerability detection,” in *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, ICSE 2024, Lisbon, Portugal, 2024*, pp. 16:1–16:13, ACM, 2024.

[9] R. Ferenc, P. Hegedüs, P. Gyimesi, G. Antal, D. Bán, and T. Gyimóthy, “Challenging machine learning algorithms in predicting vulnerable javascript functions,” in *Proceedings of the 7th International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering, RAISE@ICSE 2019, Montreal, QC, Canada, 2019*, pp. 8–14, IEEE / ACM, 2019.

[10] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov, “Roberta: A robustly optimized bert pretraining approach,” 2019.

[11] X. Zhou, T. Zhang, and D. Lo, “Large language model for vulnerability detection: Emerging results and future directions,” in *Proceedings of the 2024 ACM/IEEE 44th International Conference on Software Engineering: New Ideas and Emerging Results*, ICSE-NIER’24, (New York, NY, USA), p. 47–51, Association for Computing Machinery, 2024.

[12] “Cvefixes dataset.” https://www.kaggle.com/datasets/girish17019/cvefixes-vulnerable-and-fixed-code.