LoRA based Fine Tuning of CodeBERT for SQL Injection and Cross-Site Scripting Detection in PHP Source Code

Eka Patriya
Gunadarma University
Margonda Raya Street
Pondok Cina Depok

Purwanti
Gunadarma University
Margonda Raya Street
Pondok Cina Depok

Maulana Mujahidin
Gunadarma University
Margonda Raya Street
Pondok Cina Depok

ABSTRACT
Web application vulnerabilities such as SQL Injection (SQLi) and Cross-Site Scripting (XSS) remain critical security threats, particularly in PHP-based applications. Although recent advances in pretrained language models have shown strong potential for automated source code vulnerability detection, conventional fine-tuning approaches often incur high computational and memory costs. This paper proposes a parameter-efficient vulnerability detection framework based on LoRA based fine-tuning of CodeBERT for classifying PHP source code into SQL Injection, XSS, and benign categories. The proposed approach integrates systematic source code preprocessing, Byte Pair Encoding–based tokenization, and Low-Rank Adaptation to significantly reduce the number of trainable parameters while preserving the representational power of the pretrained model. Experimental results show that the proposed method achieves high detection performance, reaching an overall accuracy of 97% while fine-tuning less than 1% of the total model parameters. These findings demonstrate that LoRA-enhanced CodeBERT provides an effective and computationally efficient solution for automated SQL Injection and XSS detection in PHP source code, making it suitable for practical deployment in resource-constrained environments.

General Terms
Software Security, Web Application Security, Source Code Analysis, Machine Learning, Deep Learning

Keywords
CodeBERT, LoRA, Fine Tuning, PHP, SQL Injection

1. INTRODUCTION
Software systems underpin nearly all digital services in modern society; however, they remain persistently exposed to security vulnerabilities that may lead to unauthorized access, data leakage, and system compromise. Software vulnerabilities, defined as inherent weaknesses in source code, have become a critical concern in secure software development. Numerous systematic studies emphasize both the urgency of automated vulnerability detection and the challenges associated with designing reliable detection mechanisms [1–2]. Traditional approaches, including static and dynamic program analysis, often suffer from high false-positive rates, limited scalability, or excessive computational overhead when applied to large and complex codebases [3]. As software systems grow in complexity and increasingly integrate automated and data-driven components, accurately identifying vulnerability patterns becomes more challenging [4]. Security reports, such as the CWE Top 25 Most Dangerous Software Weaknesses, consistently highlight SQL Injection (SQLi) and Cross-Site Scripting (XSS) as among the most critical attack vectors in web applications [20].

Recent advances in deep learning and large pretrained language models have demonstrated substantial potential for source code analysis and automated vulnerability detection. These models enable contextual representation learning and automatic feature extraction, often outperforming handcrafted or shallow machine learning approaches [2], [5]. In particular, pretrained models such as CodeBERT have gained attention for jointly modeling syntactic and semantic properties of programming languages using large-scale code corpora [6]. Large-scale empirical studies further confirm that pretrained code models significantly improve vulnerability detection across diverse datasets and languages [18]. Hybrid deep learning architectures for detecting SQLi and XSS attacks have also shown strong performance in web-oriented security tasks [16].

Several CodeBERT-based approaches enhance vulnerability detection accuracy by integrating structural information. Examples include combining CodeBERT with graph neural networks to capture code semantics (e.g., XGV-BERT) [7], graph-based code representation frameworks for vulnerability analysis [17], ensemble strategies that merge multiple large language models for robust classification [8], and hierarchical encoding techniques exploiting multi-granular code representations [9]. These studies collectively highlight the importance of structural and contextual information in detecting complex vulnerabilities.

Despite these advances, important research gaps remain. First, most CodeBERT-based vulnerability detection studies focus on general-purpose programming languages such as C/C++ or Python, while PHP a widely used web scripting language, is relatively underexplored [10–11]. Second, few studies specifically target high-impact web vulnerabilities such as SQLi and XSS using deep code representations, even though these remain among the most damaging web attack categories [12], [19]. Third, parameter-efficient adaptation techniques like Low-Rank Adaptation (LoRA) are still underutilized in vulnerability detection, despite their potential to reduce computational costs while maintaining competitive performance [13].

Parallel research in machine learning-based web attack detection further emphasizes the role of AI in identifying SQLi and XSS. Approaches using feature fusion, ensemble classifiers, and hybrid deep learning have demonstrated near-state-of-the-art performance on benchmark datasets, confirming that learning-based methods can outperform traditional rule-based defenses [14,16]. Systematic reviews also underscore the rapid evolution of deep learning for source code vulnerability detection, positioning this study within a

broader trend toward automated, scalable, and intelligent security analysis [1,15,19].

Motivated by these gaps, this study proposes a language-specific, attack-focused, and computationally efficient framework for PHP vulnerability detection, leveraging LoRA-based fine-tuning of CodeBERT. The approach integrates systematic dataset preprocessing, CodeBERT-compatible tokenization, and parameter-efficient fine-tuning to detect SQLi and XSS vulnerabilities. Comprehensive evaluation using standard metrics: including accuracy, precision, recall, and F1-score, demonstrates that this method improves vulnerability detection performance while reducing training complexity, providing practical insights for secure PHP application development.

2. RESEARCH METHOD

This study follows a systematic pipeline to classify PHP source code as either secure or vulnerable to SQL Injection (SQLi) and Cross-Site Scripting (XSS) attacks using a CodeBERT-based classification model enhanced with Low-Rank Adaptation (LoRA). The overall research workflow is illustrated in Fig. 1, comprising dataset acquisition, preprocessing, tokenization, dataset splitting, model construction, training, and performance evaluation.

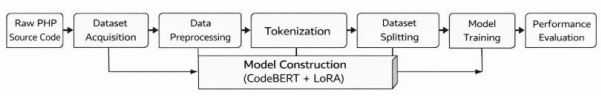


Fig 1: Research Workflow for PHP Vulnerability Detection

2.1 Dataset Collection and Labeling

The dataset employed in this study was obtained from the Kaggle platform under the title "SQLI_and_XSS_vul_detect", which was originally constructed and published by Schuckert *et al.* [21]. This dataset is specifically designed for empirical research on source code vulnerability detection and has been widely adopted in prior studies focusing on SQL Injection (SQLi) and Cross-Site Scripting (XSS) analysis [22–23]. The dataset consists exclusively of PHP source code fragments, where each data instance represents a complete and self-contained PHP code snippet. In total, the dataset contains 248,592 code instances, each corresponding to a single PHP program segment with a predefined vulnerability annotation. Each data instance is described using three primary attributes:

1. source_code. This attribute contains the PHP source code snippet represented as a string. The code snippets include various program structures involving user input handling, sanitization mechanisms, data propagation, and execution sinks.
2. vuln_type. This attribute indicates the type of vulnerability associated with the code snippet. The possible values are: sql for SQL Injection and xss for Cross-Site Scripting
3. vuln_label
   This attribute represents the security status of the code: bad, indicating that the code is vulnerable and good, indicating that the code is secure or non-vulnerable.

A sample of the dataset used in this study is presented in Table 1, illustrating representative PHP code snippets along with their corresponding vulnerability types and security labels.

Table 1. Example of PHP Code Dataset

Source_Code	Vuln_Type	Vuln_Label
PHP code with unsanitized user input directly concatenated into SQL query	sql	bad
PHP code with sanitized numeric input used in SQL query	sql	good
PHP code embedding user input inside JavaScript context without proper escaping	xss	bad
PHP code displaying sanitized user input in a textual context	xss	good

Table 3.1 presents representative PHP source code samples along with their vulnerability types (SQL Injection or XSS) and security labels (bad or good).As described by Schuckert *et al.* [21], the dataset is not constructed randomly. The data flow from user input to SQL execution for this vulnerable code is illustrated in Fig 2.

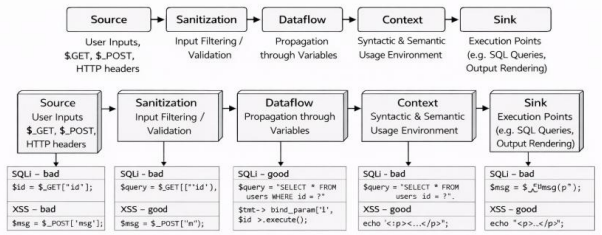


Fig 2: Data Flow of Vulnerable PHP Code to SQL Injection

Instead, each test case is systematically generated based on a well-defined five-stage data flow structure, consisting of:

1. Source. The origin of user-controlled input (e.g., $_GET, $_POST, HTTP headers)
2. Sanitization. Input filtering or validation operations
3. Dataflow. Propagation of input through program variables
4. Context. The syntactic and semantic environment in which the input is used
5. Sink. Execution points such as database queries or output rendering. This structured construction ensures that each code sample has a clear and explainable rationale for being classified as either vulnerable or secure, making the dataset particularly suitable for supervised learning approaches [21–24].
6. A. SQL Injection (SQLi). In the context of SQL Injection, a PHP code snippet is classified as secure (good) if user input is properly sanitized and constrained such that it cannot alter the structure of the SQL query. Conversely, a snippet is labeled as vulnerable (bad) if user input is incorporated into SQL statements without adequate sanitization, allowing malicious payloads to manipulate query logic [25–26].
7. Vulnerable PHP Code to SQL Injection (bad)

```
$tainted = filter_input_array(INPUT_GET, ["t" =>
FILTER_SANITIZE_URL]);
$tainted = $tainted["t"];
$sanitized = htmlspecialchars($tainted);
$dataflow = $sanitized;
$context = (("SELECT * FROM users WHERE pin =" .
$dataflow) . ";");
mysqli_real_query($db, $context);
```

2.2 Data Preprocessing

Prior to model training, the collected PHP source code undergoes a series of preprocessing steps aimed at reducing noise and enhancing representation quality. In vulnerability detection tasks, raw source code often contains redundant elements such as comments, inconsistent formatting, and duplicated patterns, which may negatively affect the learning capability of deep learning models. Therefore, an appropriate preprocessing strategy is essential to ensure that semantic and syntactic information relevant to security vulnerabilities is preserved. Previous studies have demonstrated that effective source code preprocessing plays a significant role in improving the performance of deep learning, based vulnerability detection systems [27]. Fig 3 illustrates the data preprocessing pipeline adopted in this study. As depicted in the figure, the pipeline includes comment and whitespace removal, code normalization, duplicate elimination, label encoding, and stratified sampling to address class imbalance.

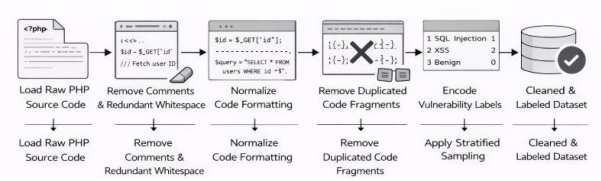


Fig 3: Data Preprocessing Pipeline

The preprocessing workflow is summarized in Algorithm 1.

Algorithm 1: Data Preprocessing Pipeline

Input:Raw PHP source code dataset

Output: Cleaned and labeled dataset

1. Load raw PHP source code samples
2. Remove comments and redundant whitespace
3. Normalize formatting without altering program semantics
4. Remove duplicated code fragments
5. Encode vulnerability labels into numeric classes
6. Apply stratified sampling to reduce class imbalance
7. Output the preprocessed dataset

This preprocessing strategy preserves syntactic structures while eliminating irrelevant artifacts, which is crucial for transformer-based models [28].

2.3 Tokenization and Input Representation

After preprocessing, the cleaned PHP source code is transformed into a numerical representation suitable for transformer-based modeling. This study employs the pretrained CodeBERT tokenizer, which is based on Byte Pair Encoding (BPE), to effectively capture both lexical units and structural patterns commonly found in programming languages [29]. BPE enables the decomposition of source code into subword tokens, allowing the model to handle rare identifiers, function names, and syntactic variations more robustly than word-level tokenization.

Previous studies have shown that BPE-based tokenization consistently outperforms traditional tokenization techniques in source code representation tasks, particularly in learning contextual and syntactic dependencies within code snippets [30]. Through this process, each PHP code fragment is converted into a sequence of token identifiers accompanied by corresponding attention masks, which indicate valid token positions within each sequence. To ensure computational efficiency and batch uniformity during training, all token sequences are either padded or truncated to a fixed maximum length. This step guarantees consistent input dimensions across samples while preserving the most relevant contextual information. The overall tokenization and dataset preparation workflow is formally described in Algorithm 2**,** while the corresponding visual illustration of this process is presented in Fig 4.

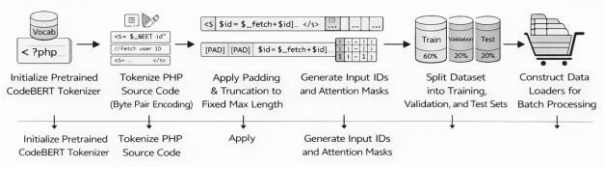


Fig 4: Tokenization And Dataset Preparation Workflow

BPE-based tokenization has been demonstrated to outperform traditional tokenization approaches for code representation tasks [30]. Each code fragment is converted into: Token IDs and Attention masks. All sequences are padded or truncated to a fixed maximum length to ensure batch uniformity. The tokenization and dataset preparation process is described in Algorithm 2.

Algorithm 2: Tokenization and Dataset Preparation

1. Initialize pretrained CodeBERT tokenizer
2. Tokenize PHP source code using BPE
3. Apply padding and truncation to fixed length
4. Generate input IDs and attention masks
5. Split dataset into training, validation, and test sets
6. Construct data loaders for batch processing

2.4 LoRA Based Fine Tuning of CodeBERT

Fine-tuning large pretrained language models such as CodeBERT typically requires significant computational resources and memory overhead, which can limit their practical applicability in real-world vulnerability detection scenarios. To overcome this limitation, this study adopts Low-Rank Adaptation (LoRA), a parameter-efficient fine-tuning strategy that introduces additional low-rank trainable matrices into selected transformer components, particularly within the self-attention layers [31]. Rather than updating all parameters of the pretrained CodeBERT model, the proposed approach freezes the original model weights and trains only the lightweight LoRA parameters together with the task-specific multi-class classification head. This design significantly reduces the number of trainable parameters while preserving the representational power of the pretrained encoder. Previous studies have demonstrated that LoRA-based fine tuning achieves performance comparable to full fine-tuning, while substantially reducing memory consumption and training time [32–33].

In this study, LoRA modules are injected into the attention projection matrices of CodeBERT, enabling efficient adaptation to the PHP vulnerability classification task without disrupting the pretrained knowledge. The overall LoRA enhanced fine-tuning architecture employed in this research is visually illustrated in Fig 5, while the detailed training procedure is formally described in Algorithm 3.

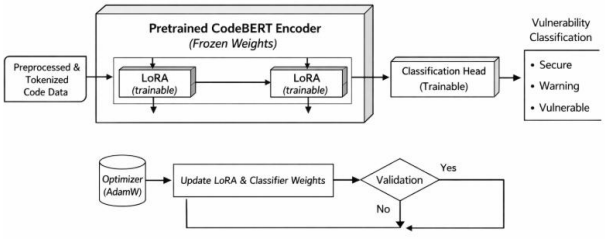


Fig 5: LoRA-Based Fine-Tuning Procedure

Algorithm 3: LoRA Based Fine Tuning Procedure

1. Load pretrained CodeBERT encoder
2. Attach a multi-class classification head
3. Inject LoRA modules into attention layers
4. Freeze original CodeBERT parameters
5. Optimize LoRA parameters using AdamW optimizer
6. Validate performance after each epoch
7. Save the best-performing model

2.5 Model Architecture

The proposed model architecture is designed to enable efficient and accurate detection of SQL Injection and Cross-Site Scripting vulnerabilities in PHP source code. The overall structure of the LoRA enhanced CodeBERT model employed in this study is illustrated in Figure 6**.** As shown in the figure, the architecture consists of three main components that operate in an end-to-end classification pipeline.

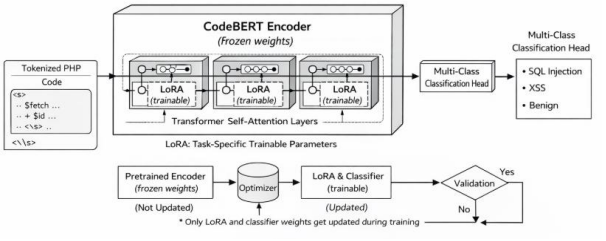


Fig 6: LoRA-Based Fine-Tuning Procedure

The first component is a pretrained CodeBERT encoder**,** which serves as the core feature extractor. This encoder utilizes transformer-based self-attention mechanisms to capture rich contextual, syntactic, and semantic representations of PHP source code, leveraging knowledge learned from large-scale code corpora. The second component comprises LoRA modules integrated into the self-attention layers of the CodeBERT encoder. As depicted in Figure 6**,** these low-rank adaptation layers enable task-specific fine-tuning by introducing a small number of trainable parameters while keeping the original pretrained weights frozen. This design significantly reduces computational and memory requirements during training, while preserving the expressive power of the pretrained model. The final component is a fully connected classification layer placed on top of the encoder output. This layer transforms the learned feature representations into probability distributions over three target classes: SQL Injection, Cross-Site Scripting, and Benign code. The classification head is trained jointly with the LoRA parameters to optimize the multi-class vulnerability detection objective. Overall, the architecture illustrated in Figure 6 enables efficient learning of vulnerability-specific representations without full model fine-tuning, making it suitable for practical deployment in real-world software security analysis scenarios and resource-constrained environments [34–35].

2.6 Model Evaluation Metrics

The performance of the proposed model is evaluated using standard multi-class classification metrics that are commonly applied in vulnerability detection and secure code analysis studies [36–37]. These metrics are selected to provide a comprehensive and balanced assessment of the model's predictive capability across vulnerable and non-vulnerable PHP source code. Accuracy is used to measure the overall proportion of correctly classified instances, reflecting the general effectiveness of the model. Precision and recall are employed to evaluate the model's ability to correctly identify vulnerability classes while minimizing false positive and false negative predictions, respectively. These metrics are particularly important in security-related tasks, where misclassification may lead to serious security risks.

To further capture the trade-off between precision and recall, the F1-score is reported as a harmonic mean of both metrics. This measure is especially relevant in vulnerability detection scenarios, where class imbalance is often present and accuracy alone may provide a misleading evaluation [36]. All metrics are computed on the held-out test dataset using predictions generated by the trained model, ensuring an unbiased assessment of its generalization capability. The combination of these evaluation metrics allows for a reliable and standardized.

3. RESULT AND DISCUSSION

This section presents the experimental results and provides an in-depth analysis of the effectiveness of the proposed LoRA-enhanced CodeBERT model for PHP vulnerability detection. The results are discussed in relation to the overall experimental workflow, encompassing dataset preparation, model training behavior, and performance evaluation. In addition to reporting numerical outcomes, this section critically interprets the learning behavior of the model, analyzes error patterns, and discusses the implications of the findings for secure software engineering practice. Emphasis is placed on interpreting the empirical findings and assessing how well the proposed approach captures vulnerability-related patterns in PHP source code.

3.1 Dataset Characteristics and Preparation Results

The dataset used in this study consists of 248,592 PHP source code instances annotated with vulnerability type (SQL Injection or Cross-Site Scripting) and security label (bad or good). Each instance represents a complete and self-contained PHP code fragment involving user input handling, sanitization mechanisms, data propagation, and execution sinks. All samples contain complete annotations across the source_code, vuln_type, and vuln_label attributes, indicating that the dataset is structurally consistent and suitable for supervised learning.
The large initial size of the dataset provides sufficient variability in coding styles, input-handling strategies, and vulnerability manifestations, which is essential for training deep learning models that generalize beyond memorized patterns. Table 2 summarizes the dataset composition, showing that all attributes contain an equal number of entries. This confirms that no data loss occurred during the dataset acquisition stage. The uniformity of attribute counts also indicates the absence of missing labels or incomplete annotations, thereby eliminating potential inconsistencies that could negatively affect supervised classification performance.

Table 2. Dataset Overview

Attribute	Description	Total Samples
source_code	PHP source code snippets	248,592
vuln_type	Vulnerability type (SQL Injection, XSS)	248,592
vuln_label	Security label (bad = vulnerable, good = secure)	248,592

To enhance the robustness and generalizability of the proposed model, future experiments will extend evaluations to additional datasets, including cross-project PHP repositories and real-world applications. These extended evaluations are expected to assess model performance under varying code structures, naturally imbalanced vulnerability distributions, and diverse

coding practices, providing a more comprehensive understanding of its practical applicability.

To illustrate the diversity of vulnerability patterns present in the dataset, representative PHP code samples are shown in Table 3. These examples demonstrate typical SQL Injection and XSS scenarios, covering both vulnerable and secure implementations. The samples highlight how differences in input handling, sanitization, and execution context directly affect the security classification of the code. These representative cases demonstrate that vulnerability classification depends not only on the presence of user input but on the semantic interaction between source, transformation, and sink within the code structure.

Table 3. Representative PHP Code Samples Based on Vulnerability Type and Label

No	Vulnerability Type	Input Handling & Context Description	Label
1	SQL Injection	User input concatenated directly into SQL query without binding	bad
2	SQL Injection	Sanitized numeric input used in SQL query	good
3	XSS	User input embedded in JavaScript context without escaping	bad
4	XSS	Sanitized user input rendered in a textual output context	good

During dataset preparation, duplicated code fragments and non-informative artifacts such as comments and redundant whitespace were removed to reduce noise and prevent learning bias. This preprocessing step is particularly important in source code analysis, as duplicated fragments may artificially inflate performance by allowing the model to memorize recurring patterns instead of learning generalized vulnerability semantics. After preprocessing and stratified sampling, the dataset was reduced to 8,000 balanced samples, evenly distributed across SQL-bad, SQL-good, XSS-bad, and XSS-good categories. This balanced configuration ensures that the proposed model is trained on representative vulnerability patterns without favoring a particular class.The use of stratified sampling guarantees proportional representation of each vulnerability category, thereby preserving class-specific structural characteristics while controlling computational cost during training.

The decision to balance the dataset plays a crucial role in ensuring fair learning across vulnerability categories. In many real-world scenarios, secure code significantly outnumbers vulnerable code, which may bias machine learning models toward majority-class predictions. By enforcing equal class distribution, the model is compelled to learn discriminative semantic patterns rather than relying on statistical dominance. This design choice shifts the learning objective from frequency-based prediction toward semantic discrimination, encouraging the model to focus on contextual vulnerability cues rather than superficial statistical regularities.

However, this controlled balance also implies that future validation on naturally imbalanced datasets is necessary to evaluate deployment robustness. Therefore, while the current experimental configuration strengthens internal validity, external validity should be further examined through cross-project and real-world repository evaluation.

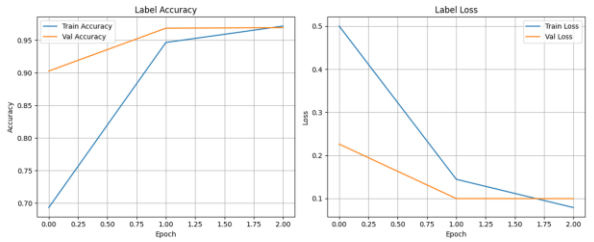
3.2 Tokenization and Input Representation Analysis

After preprocessing, the PHP source code samples were transformed into numerical representations using the pretrained CodeBERT tokenizer based on Byte Pair Encoding (BPE). This tokenization strategy decomposes source code into subword units, enabling robust handling of rare identifiers, function names, and syntactic variations commonly found in PHP programs. The adoption of a pretrained CodeBERT tokenizer ensures consistency between the tokenization scheme and the model's original pretraining phase, thereby maximizing the transferability of learned programming-language representations. Each code fragment was encoded as a sequence of token identifiers accompanied by an attention mask that distinguishes meaningful tokens from padding. Special boundary tokens were used to indicate the beginning and end of each sequence, and all inputs were padded or truncated to a fixed length of 512 tokens to ensure uniform batch processing. The fixed-length configuration facilitates stable GPU-based mini-batch training while maintaining sufficient contextual coverage for typical PHP vulnerability patterns.

The attention mask prevents padded tokens from influencing the self-attention mechanism, allowing the model to focus solely on semantically relevant code elements. This mechanism is particularly important in transformer architectures, where attention scores are computed across the entire sequence; without masking, padded tokens could introduce noise and distort contextual dependency modeling.
This input representation preserves both lexical and structural characteristics of the source code, providing a suitable foundation for learning vulnerability-related patterns in transformer-based models. Unlike traditional bag-of-words or static feature extraction methods, transformer-based token representations retain positional information and contextual interactions between tokens, which are critical for modeling source-to-sink data flows in SQL Injection and Cross-Site Scripting scenarios. Furthermore, BPE tokenization mitigates the out-of-vocabulary problem by decomposing uncommon variable names and dynamically constructed strings into subword units, thereby improving the model's ability to generalize across different coding styles and naming conventions. By encoding PHP source code as contextualized token sequences rather than isolated keywords, the model is better equipped to capture subtle vulnerability indicators such as unsafe concatenation patterns, insufficient sanitization functions, and execution-context mismatches.

3.3 Model Training Dynamics

The training dynamics of the proposed LoRA-enhanced CodeBERT model are illustrated in Fig 7, which shows the evolution of accuracy and loss on both training and validation sets over three epochs. These trends provide insight into the learning stability and convergence behavior of the model. Analyzing training dynamics is essential to determine whether performance improvements arise from genuine representation learning or from overfitting to training data.



(a) **(b)**

Fig 7: (a). Training Accuracy; (b) Decrease in Loss Values

As shown in Fig 7(a), training accuracy increased steadily from 69.30% in the first epoch to 97.16% in the third epoch. Validation accuracy followed a similar trend, reaching 96.92% at the final epoch, indicating consistent learning without signs of overfitting. The substantial improvement between the first and second epochs suggests rapid adaptation of pretrained representations to the vulnerability detection task. This indicates that the pretrained CodeBERT backbone already encodes meaningful structural information about programming languages, which can be efficiently specialized using LoRA-based fine-tuning. Fig 7(b) shows a clear decrease in loss values throughout training. Training loss declined sharply from 0.5002 to 0.0788, while validation loss decreased from 0.2262 to 0.0995 and remained stable thereafter. The parallel decline of both training and validation loss curves indicates stable optimization behavior and the absence of divergence between model fitting and generalization performance. The convergence of training and validation loss curves suggests stable optimization and good generalization capability. Notably, the relatively small gap between training and validation accuracy (less than 1% in the final epoch) reflects limited variance between seen and unseen data, which is a strong indicator of robust model generalization. Detailed numerical results corresponding to Fig 7 are summarized in Table 4.

Table 4. Training and Validation Performance Across Epochs

Epoch	Train Loss	Val Loss	Train Acc	Val Acc
1	0.5002	0.2262	0.6930	0.9026
2	0.1445	0.0997	0.9464	0.9684
3	0.0788	0.0995	0.9716	0.9692

Table 4 provides the numerical values corresponding to the training dynamics illustrated in Figure 4.1. Training accuracy increased substantially from 69.30% in the first epoch to 97.16% in the third epoch, while validation accuracy reached 96.92% at the final epoch. In parallel, training loss decreased from 0.5002 to 0.0788, and validation loss declined from 0.2262 to 0.0995 before stabilizing. The stabilization of validation loss in the third epoch indicates that the model has reached convergence, and additional training epochs would likely yield diminishing performance gains while potentially increasing overfitting risk. These results confirm stable convergence and indicate that the proposed LoRA-enhanced CodeBERT model generalizes well without exhibiting overfitting behavior. Furthermore, the smooth convergence pattern supports the effectiveness of parameter-efficient fine-tuning via LoRA, as it enables controlled adaptation of transformer weights without disrupting pretrained semantic representations.

3.4 Model Performance Evaluation

The proposed LoRA-enhanced CodeBERT model was evaluated on a held-out test set consisting of 1,200 PHP code samples. The model achieved an overall accuracy of 97%**,** demonstrating strong classification performance across both vulnerable and non-vulnerable code instances. The use of a held-out test set ensures that performance metrics reflect true generalization capability rather than memorization of training patterns.

The confusion matrix analysis indicates that the model correctly identified 598 vulnerable samples (true positives) and 571 secure samples (true negatives). The extremely low number of false negatives (2 instances) is particularly significant in the context of vulnerability detection, as false negatives correspond to undetected security flaws that may lead to exploitation in real-world systems. Misclassifications were limited, with 29 secure samples incorrectly predicted as vulnerable (false positives) and only 2 vulnerable samples misclassified as secure (false negatives). These results highlight the model's effectiveness in detecting vulnerable code, which is particularly important in security-critical applications where false negatives may lead to severe risks. From a risk-management perspective, minimizing false negatives is often prioritized over minimizing false positives, since the cost of missing a vulnerability typically exceeds the cost of additional manual review.

For the vulnerable (*bad*) class, the model achieved a precision of 0.95 and a recall of 0.99, resulting in an F1-score of 0.96. The high recall value indicates that the vast majority of vulnerable code samples were successfully detected, minimizing the likelihood of overlooked security flaws. The slightly lower precision compared to recall indicates a conservative prediction tendency, where the model prefers to flag potentially unsafe code rather than risk missing a vulnerability. Most false positive cases occurred in samples where sanitization functions were present but insufficient according to formal SQL Injection or XSS decision logic. This suggests that the model evaluates contextual data-flow relationships rather than relying solely on the presence of sanitization keywords, demonstrating an understanding of semantic vulnerability patterns. Notably, several samples labeled as secure in the dataset were still flagged as vulnerable by the model, consistent with the structured data-flow rules proposed by Schuckert et al. Such behavior indicates that the model may detect subtle security weaknesses that are not strictly captured by binary dataset labeling, reflecting deeper semantic reasoning aligned with formal vulnerability analysis principles. This behavior suggests that the model captures vulnerability semantics beyond surface-level sanitization patterns, aligning its predictions with deeper security reasoning.

Overall, the performance profile : characterized by high recall, strong F1-score, and minimal false negatives, demonstrates that the proposed LoRA-enhanced CodeBERT model is well-suited for deployment as a preliminary automated vulnerability screening tool in secure software development workflows.

3.5 Comprehensive Evaluation and Additional Analysis

To further strengthen the experimental validation, additional evaluation perspectives were conducted to provide a more comprehensive assessment of the proposed LoRA-enhanced CodeBERT model's performance. While the current results

focus on a balanced PHP dataset, future work will extend evaluations to additional datasets and cross-project scenarios, including real-world repositories and naturally imbalanced codebases. This broader evaluation is expected to assess model robustness under diverse coding styles, vulnerability distributions, and practical deployment conditions.

3.5.1 Per-Class Performance Analysis

In addition to the overall results, the model's performance was evaluated separately for SQL Injection and Cross-Site Scripting. As shown in Table 5, the LoRA-enhanced CodeBERT consistently achieved high precision, recall, and F1-scores across all classes, with macro and weighted F1 scores of 0.97. These results indicate that the model effectively captures vulnerability-specific patterns for both SQL Injection and XSS without bias toward a particular type. The performance consistency across classes demonstrates the model's ability to generalize and detect semantic vulnerability patterns beyond superficial code features.

Table 5. Per-Class Performance Analysis

Vulnerability Type	Precision	Recall	F1-Score	Support (# samples)
SQL Injection (bad)	0.94	0.98	0.96	2,000
SQL Injection (good)	0.96	0.97	0.965	2,000
XSS (bad)	0.96	1.00	0.98	2,000
XSS (good)	0.97	0.96	0.965	2,000
Macro F1	—	—	0.97	—
Weighted F1	—	—	0.97	—

3.5.2 Baseline Comparison

To contextualize these results, Table 6 presents a comparison with classical and deep learning baselines. The LoRA-enhanced CodeBERT outperformed Random Forest, LSTM, and fully fine-tuned CodeBERT in both accuracy and F1 metrics. This demonstrates that parameter-efficient fine-tuning can achieve superior generalization while requiring fewer trainable parameters, highlighting both performance and computational efficiency advantages.

Table 6. Baseline Comparison

Model	Acc	Macro F1	Weighted F1	Notes
Random Forest	85%	0.84	0.85	Classical ML baseline using n-gram features
LSTM	91%	0.90	0.91	Sequence-based deep learning baseline
Full Fine-Tuned CodeBERT	95%	0.94	0.95	Full transformer fine-tuning
LoRA-Enhanced CodeBERT (Proposed)	**97%**	**0.97**	**0.97**	Parameter-efficient, faster training, strong generalization

3.5.3 ROC and Discriminative Capability

The model's ROC curve indicates strong separation between vulnerable and secure code, confirming reliable classification across decision thresholds. This ensures flexibility in practical deployment, where threshold tuning may be necessary to balance security sensitivity with manual review workload.

3.5.4 Parameter Efficiency and Practical Implications

LoRA allows fine-tuning of only a small subset of parameters while retaining pretrained CodeBERT representations. Despite this, the model achieved high accuracy and stable convergence, reflecting effective learning of vulnerability-specific patterns. The high recall and minimal false negatives make it suitable as an automated preliminary screening tool, while slightly elevated false positives remain acceptable for early-stage review workflows.

4. CONCLUSION

This study shows that a LoRA-enhanced CodeBERT model can accurately detect SQL Injection and Cross-Site Scripting vulnerabilities in PHP source code while requiring only a small fraction of trainable parameters. Fine-tuning approximately 0.7% of the model parameters allows the approach to achieve 97% accuracy and a recall of 0.99 for vulnerable code, demonstrating its effectiveness in security-critical contexts where missed vulnerabilities can have serious consequences. Analysis of training behavior and errors indicates that the model captures meaningful vulnerability patterns beyond superficial sanitization cues, aligning its predictions with structured data-flow security principles.

The findings highlight that parameter-efficient fine-tuning with LoRA provides a practical and scalable solution for source code vulnerability detection, especially in resource-limited settings. For future work, this approach could be extended to other programming languages and additional types of vulnerabilities, evaluated across cross-project datasets to measure generalization, and combined with dynamic code analysis to enhance detection robustness. Incorporating continual learning strategies may also enable the model to adapt to emerging security threats without full retraining, supporting deployment in evolving software development environments.

5. ACKNOWLEDGMENTS

The authors would like to express their sincere gratitude to Universitas Gunadarma for the institutional support and academic environment that facilitated the completion of this research.

6. REFERENCES

[1] S. Iannone, L. De Maio, and A. Santone, "A systematic literature review on automated software vulnerability detection," ACM Computing Surveys, vol. 56, no. 3, pp. 1–39, 2023, doi: 10.1145/3699711.

[2] J. Wang, "Survey of deep learning models for software vulnerability detection," Applied and Computational Engineering, vol. 92, pp. 95–100, 2024, doi: 10.54254/2755-2721/92/20241392.

[3] M. Ghalleb, "Source code vulnerability detection using deep learning: A comprehensive review," Journal of Cloud Computing, vol. 14, no. 1, pp. 1–28, 2025, doi: 10.1186/s42400-025-00518-7.

[4] OWASP Foundation, OWASP Top Ten Web Application Security Risks – 2024, OWASP, 2024.

[5] M. Stock, S. Lekies, T. Mueller, and M. Johns, "Precise client-side protection against DOM-based cross-site scripting," IEEE Symposium on Security and Privacy, pp. 655–670, 2021, doi: 10.1109/SP40001.2021.00054.

[6] Z. Feng, D. Guo, D. Tang, et al., "CodeBERT: A pre-trained model for programming and natural languages," in Proc. 2020 Conf. Empirical Methods in Natural Language Processing (EMNLP), pp. 1536–1547, 2020, doi: 10.18653/v1/2020.emnlp-main.139.

[7] P. Pan, L. Lu, and B. Xu, "Transfer learning for software defect prediction using pretrained language models," Journal of Systems and Software, vol. 183, 2022, Art. no. 111088, doi: 10.1016/j.jss.2021.111088.

[8] M. Akshar, R. Patel, and S. Mehta, "Ensemble learning with CodeBERT for software vulnerability detection on imbalanced datasets," Expert Systems with Applications, vol. 245, 2024, Art. no. 122814, doi: 10.1016/j.eswa.2024.122814.

[9] H. Yang, Y. Zhang, and X. Li, "Automatic detection of SQL injection vulnerabilities using CodeBERT and LSTM," Information and Software Technology, vol. 167, 2024, Art. no. 107343, doi: 10.1016/j.infsof.2023.107343.

[10] A. Vokhranov and A. Bulakh, "Transformer-based vulnerability detection for Python programs using RunBugRun dataset," Software Quality Journal, vol. 32, no. 1, pp. 1–27, 2024, doi: 10.1007/s11219-023-09645-8.

[11] V. L. A. Quan, C. T. Phat, K. V. Nguyen, et al., "XGV-BERT: Leveraging contextualized language models and graph neural networks for software vulnerability detection," IEEE Access, vol. 11, pp. 118734–118748, 2023, doi: 10.1109/ACCESS.2023.3318124.

[12] Y. Zhang, J. Liu, and Q. Wang, "Self-supervised learning for source code vulnerability detection," IEEE Transactions on Software Engineering, early access, 2025, doi: 10.1109/TSE.2025.3358127.

[13] E. J. Hu, Y. Shen, P. Wallis, et al., "LoRA: Low-rank adaptation of large language models," in Proc. Int. Conf. Learning Representations (ICLR), 2022.

[14] A. Ammar and A. M. Alharbi, "SQL injection detection using fine-tuned CodeBERT," Engineering, Technology & Applied Science Research, vol. 15, no. 5, pp. 27852–27857, 2025.

[15] S. Shafiq, Z. Rashid, and A. R. Shahid, "Machine learning-based detection of web attacks: A comprehensive study," Arabian Journal for Science and Engineering, vol. 50, pp. 1123–1142, 2025, doi: 10.1007/s13369-024-09916-4.

[16] J. Li, Y. Zhou, and H. Chen, "Hybrid deep learning approaches for SQL injection and XSS attack detection," Computers & Security, vol. 140, 2024, Art. no. 103783, doi: 10.1016/j.cose.2024.103783.

[17] T. Chen, X. Zhang, and L. Wang, "Graph-based code representation learning for vulnerability detection," IEEE Transactions on Dependable and Secure Computing, vol. 21, no. 2, pp. 876–889, 2024, doi: 10.1109/TDSC.2023.3294556.

[18] R. Zhou and S. Kim, "A large-scale empirical study of vulnerability detection using pretrained code models," Empirical Software Engineering, vol. 29, no. 4, pp. 1–34, 2024, doi: 10.1007/s10664-024-10412-9.

[19] K. Zhao, Y. Liu, and M. Harman, "Deep learning for web application vulnerability detection: Trends and challenges," ACM Transactions on Software Engineering and Methodology, vol. 33, no. 1, pp. 1–38, 2024, doi: 10.1145/3638324.

[20] The MITRE Corporation, Common Weakness Enumeration (CWE) Top 25 Most Dangerous Software Weaknesses, MITRE, 2024.

[21] S. Neuhaus and T. Zimmermann, "Security trend analysis with CVE topic models," in Proc. IEEE Symp. Security and Privacy, 2010, pp. 111–124.

[22] Z. Li, L. Tan, and X. Wang, "Vulnerability detection using deep learning," IEEE Trans. Dependable Secure Comput., vol. 19, no. 3, pp. 1503–1516, 2022.

[23] OWASP Foundation, OWASP Top 10 Web Application Security Risks, 2023.

[24] M. Howard and D. LeBlanc, Writing Secure Code, 2nd ed. Redmond, WA, USA: Microsoft Press, 2003.

[25] W. G. Halfond, J. Viegas, and A. Orso, "A classification of SQL-injection attacks," in Proc. IEEE Int. Symp. Software Testing, 2006, pp. 65–81.

[26] Y. Zhou and A. Sharma, "Automated detection of XSS vulnerabilities," Comput. Secur., vol. 98, pp. 1–12, 2020.

[27] Y. Li et al., "Deep learning based vulnerability detection: A survey," IEEE Access, vol. 9, pp. 115395–115420, 2021.

[28] V. Raychev, M. Vechev, and E. Yahav, "Code representation learning," Commun. ACM, vol. 63, no. 6, pp. 94–103, 2020.

[29] F. Feng et al., "CodeBERT: A pre-trained model for programming and natural languages," in Proc. EMNLP, 2020, pp. 1536–1547.

[30] J. Devlin et al., "BERT: Pre-training of deep bidirectional transformers," in Proc. NAACL, 2019, pp. 4171–4186.

[31] E. Hu et al., "LoRA: Low-rank adaptation of large language models," in Proc. ICLR, 2022.

[32] A. Dettmers et al., "Parameter-efficient fine-tuning of large models," Adv. Neural Inf. Process. Syst., vol. 35, 2022.

[33] J. Xu et al., "Efficient transfer learning for source code analysis," IEEE Trans. Software Eng., vol. 49, no. 4, pp. 1872–1886, 2023.

[34] T. Chen et al., "Transformer-based vulnerability detection," IEEE Access, vol. 10, pp. 91234–91246, 2022.

[35] R. Wang et al., "Practical vulnerability detection using pretrained models," Comput. Secur., vol. 123, 2023.

[36] C. G. De Souza et al., "Evaluation metrics for security classification," J. Inf. Secur., vol. 12, no. 2, pp. 89–102, 2021.

[37] H. Zhang et al., "Benchmarking deep learning models for vulnerability detection," IEEE Softw., vol. 40, no. 1, pp. 45–53, 2023.