Proceedings of Engineering and Technology Innovation, vol. 32, 2026, pp. 28-42

Real-Time Code Vulnerability Detection Using a Machine Learning-Integrated Language Server

Ariel Roy Luceño Reyes*, Mark David Dayanan Prado, Raffy Beting Suarez,
Rovenado Nesta Abellana Villotes

College of Information and Computing, University of Southeastern Philippines, Davao, Philippines

Received 22 April 2025; received in revised form 12 November 2025; accepted 17 November 2025
DOI: https://doi.org/10.46604/peti.2025.15065

Abstract

The rapid growth of software development has improved productivity but also introduced security risks, especially when developers skip essential scans due to time constraints or limited tool support. This study proposes a real-time vulnerability detection system that integrates machine learning (ML) into a language server framework to enhance software security during coding. The system uses a Language Server Protocol (LSP) architecture with a Random Forest classifier that analyzes source code at the line level. Code is pre-processed through tokenization, abstract syntax tree (AST) traversal, and TF-IDF vectorization before being classified into four vulnerability types: CWE-79 (Cross-Site Scripting), CWE-89 (SQL Injection), CWE-22 (Path Traversal), and CWE-434 (Unrestricted File Upload). Using 20,000 labeled code lines, the model achieves 82.3% accuracy and an F1-score of 80.7%, performing best on CWE-79 and CWE-89 and showing weakest performance on CWE-434. The language server averages 72 ms per diagnostic, demonstrating its suitability for real-time developer workflows.

Keywords: code vulnerability detection, CWE, Language Server Protocol, machine learning

1. Introduction

The rapid growth of software development has revolutionized operations for both enterprises and individuals, enabling them to handle daily tasks more efficiently, accelerate processes, and enhance overall productivity. However, this shift toward digital solutions has also introduced significant security risks. According to Cabrera et al. [1], security is one of the biggest concerns in the digital world. Many software applications are developed with vulnerabilities that attackers can exploit, often resulting in data breaches, financial losses, and reputational damage [2-3]. Studies reveal that most vulnerabilities emerge during the coding phase of the software development lifecycle, often made worse by developers frequently bypassing crucial security scans due to time constraints and a lack of proper tooling [4].

Software vulnerabilities, defined as weaknesses or flaws that can be exploited to compromise a system's confidentiality, integrity, or availability, continue to pose a significant challenge in cybersecurity [5]. Common security vulnerabilities include buffer overflows and injection-based attacks, which can impact popular software applications [6-7]. For example, an SQL Injection vulnerability in Palo Alto Networks Expedition, identified as CVE-2024-9465, allowed unauthorized access to sensitive data and system functions [8]. Similarly, CVE-2024-51378 showcased the exploitation of poorly secured endpoints, allowing attackers to bypass authentication mechanisms and execute arbitrary commands [9].

* Corresponding author. E-mail address: ariel.reyes@usep.edu.ph

Despite the urgency of addressing these issues, traditional vulnerability detection tools often fail to integrate seamlessly into modern development workflows. Although effective at identifying vulnerabilities before the production phase of software products, static analysis tools are typically separate from live coding environments, resulting in prolonged remediation times and inefficiencies [10]. This reflects a broader challenge where process improvements for integrating security tools into software development methodologies have not been clearly identified, leaving organizations vulnerable to recurring flaws despite the availability of security practices and technologies [4].

The rise of machine learning (ML) has presented promising opportunities for enhancing software security. Advanced models have demonstrated high accuracy in identifying vulnerabilities, with some achieving a high F1-score in detecting issues like buffer overflows and injection attacks [6]. Recent studies have also progressed toward line-level vulnerability detection, enabling models to pinpoint the exact statements responsible for security flaws rather than labeling entire functions or files [11]. This finer granularity enhances both precision and interpretability, allowing developers to better understand the origins of vulnerabilities [12]. However, integrating these capabilities into real-time coding environments remains largely unexplored. Current research lacks comprehensive solutions that enable developers to identify vulnerabilities dynamically as they write code, a gap this study seeks to address.

This paper introduces a method for real-time line-level vulnerability detection by integrating ML into a language server. As a proof of concept, the primary goal is to demonstrate the feasibility of improving software security during development by:

(1) enabling real-time identification of vulnerabilities during the coding process; and
(2) achieving a decent balance between detection accuracy and false positives through a tailored ML model.

The goal of this work is to translate vulnerability detection theory into practical applicability within live development environments, furthering efforts to enhance software security at the point of code creation.

Section 2 reviews related work, covering research on coding practices, prevalent security weaknesses, existing detection solutions, ML approaches, and applications of the Language Server Protocol (LSP). Section 3 describes the architecture of the proposed framework, focusing on the language client, language server, and analysis component. Section 4 outlines the experimental methodology and reports the results obtained for each component, including the ML-based Analysis Component. Lastly, Section 5 summarizes the main findings and discusses potential avenues for future research.

2. Literature Review

This section examines the key factors that influence software vulnerability detection, beginning with how modern coding practices increase complexity and introduce security risks. By reviewing common weaknesses, real-world exploits, and established detection techniques, it highlights the limitations of current approaches. It also analyzes recent advances in ML and the role of the LSP in enabling real-time analysis. Through this review, the section aims to clarify the technological and methodological foundations needed to develop more accurate and efficient vulnerability detection systems.

2.1. Coding practices

The shift from procedural programming to modular and layered designs has improved scalability and reusability in software development, but has also brought new security concerns. Modularity allows faster development and easier code reuse, yet it also broadens the attack surface because vulnerabilities may be spread across interconnected components [3]. In some cases, long dependency chains can hide insecure code, making flaws more challenging to detect than in earlier procedural programs. At the same time, features that make software easier to maintain, such as abstraction and layered architecture, can

hide weaknesses from developers. Alenezi and Zarour [3] note that as systems become more complex, vulnerabilities typically increase, meaning that higher productivity benefits often come with greater security risks. For this reason, evaluating development practices only in terms of efficiency or scalability is incomplete. Secure coding today depends on finding a balance: taking advantage of modern design techniques while implementing safeguards to limit the risks that complexity introduces.

2.2. Common vulnerabilities and their exploitation

As coding practices advanced, the vulnerabilities created by complexity became more apparent. Many of these problems came from input validation and data handling mistakes, leaving applications vulnerable to exploitation. Over time, recurring flaws of this kind have been cataloged in the Common Weakness Enumeration (CWE), a comprehensive repository that tracks flaws in operational systems [13]. A well-known example is CWE-79 (Cross-Site Scripting), where poor user input handling lets malicious scripts run in a browser [14]. Reflected Cross-Site Scripting shows how this works in practice. When unvalidated inputs are echoed back in HTTP responses, attackers can craft links that execute harmful code in a user's browser [14]. This weakness is often exploited in phishing attacks, where fake websites trick users into sharing login details or verification codes [15-17]. Issues that begin with simple input handling can therefore lead to more significant social engineering threats.

Specific cases also show the consequences of insecure practices. CVE-2024-9465 exposed database contents through SQL Injection, and CVE-2024-51378 allowed attackers to bypass authentication and execute commands due to poor input handling [8-9]. These incidents demonstrate how minor mistakes during development can escalate into breaches, resulting in significant financial, operational, and reputational costs. All these point to the need for secure coding to be built into development from the start. Relying on fixes after deployment exposes systems, especially when developers are under pressure or lack security training. As software becomes increasingly complex, detecting vulnerabilities during coding is essential to break the cycle of recurring flaws.

2.3. Traditional solutions for vulnerability detection

Manual code reviews have long been a cornerstone of identifying security weaknesses in the software development lifecycle. Studies of open-source projects such as OpenSSL and PHP confirm that this practice can reveal significant vulnerabilities [18]. However, reviews often stall when developers disagree on remediation or fail to follow up, showing that human oversight alone cannot reliably enforce security. This shortcoming has led to the adoption of static code analysis as a complementary approach, where vulnerabilities are detected without executing the code [19].

Static analysis offers a clear advantage over manual reviews in terms of scalability and speed, particularly in detecting recurring patterns of insecure code [10]. Tools such as Joern extend this benefit by combining structural analysis with metrics to highlight weaknesses and measure code quality [7, 10]. However, these methods are not without drawbacks. Unlike manual reviews, which can incorporate contextual judgment, static tools often generate false positives [20], thereby reducing trust in their results. This tension highlights a trade-off: manual reviews provide nuanced insight but are limited in coverage and consistency, while static tools broaden detection but risk eroding developer confidence. For effective vulnerability management, the literature suggests that neither approach is sufficient in isolation; combining manual expertise with automated detection offers a more balanced path forward.

2.4. Machine learning in vulnerability detection

ML has increasingly been recognized as a viable alternative to traditional vulnerability detection methods [20]. While static analysis remains widely used, it struggles with complex or non-obvious code patterns. ML models, in contrast, have demonstrated stronger performance against common attack vectors, such as buffer overflow and injection-based exploits. Deep

learning techniques, including graph neural networks and token-based embeddings, have achieved F1-scores approaching 95% on benchmark datasets [6]. These findings represent a significant advancement beyond the limitations of earlier static approaches, particularly for large and diverse code repositories.

Recent studies have increasingly focused on line-level vulnerability detection, which pinpoints the exact lines of code responsible for security flaws. Fu and Tantithamthavorn [11] introduced LineVul, a Transformer-based model that uses contextual embeddings to improve detection accuracy, achieving up to 379% higher F1-scores compared to baseline models. Similarly, Hin et al. [12] proposed LineVD, which integrates both control and data dependency information to identify vulnerable lines while scanning less than half of the total codebase. Together, these works highlight the shift toward more fine-grained and interpretable detection approaches that improve precision and reduce manual inspection costs.

Despite these advancements, most transformer-based approaches, including those designed for line-level analysis, remain computationally demanding and often unsuitable for real-time or resource-constrained environments [21]. In contrast, traditional ML classifiers continue to provide faster inference, lower resource requirements, and greater interpretability. Comparative studies have reinforced this finding. For example, Rajapaksha et al. [22] evaluated multiple classifiers on C/C++ programs and found that Random Forest achieved the highest binary classification F1-score (0.96) with minimal variance across folds. At the same time, XGBoost performed best in multi-class scenarios with an F1-score of 0.85.

Similarly, Yang and Wang [23] compared twelve ML models for industrial failure prediction and reported that Random Forest attained 99.5% accuracy, with equally high recall (99.5%) and F1-score (99.39%), significantly outperforming Support Vector Machine (77.93%) and Logistic Regression (77.57%). Although Decision Tree and XGBoost also demonstrated strong performance, they required extensive tuning and were more prone to overfitting or higher computational cost. Supporting these results, Zaharia et al. [24] evaluated several classifiers for detecting vulnerabilities in source code. They found that Random Forest achieved the best recall (up to 0.899) for C/C++ code, showing strong resistance to noise and data imbalance. Collectively, these findings highlight Random Forest's consistent accuracy and operational efficiency, driven by its ensemble-based mechanism that minimizes the need for complex parameter optimization.

Overall, Random Forest offers a balanced combination of accuracy, computational efficiency, and interpretability for real-time vulnerability detection. Its ensemble structure reduces the overfitting issues common in single Decision Trees, while its parallel nature enables faster inference compared to deep learning models, including Transformer-based detectors. This combination of reliability, scalability, and low operational overhead makes Random Forest a practical choice for integration into development environments and continuous integration pipelines.

2.5. Language Server Protocol (LSP)

The LSP has revolutionized the integration of language-specific tools into development environments. By establishing a standardized protocol, LSP simplifies the development of features like syntax highlighting, error detection, and auto-completion across various programming languages [25]. This standardization reduces the complexity of providing language support for multiple editors, allowing developers to focus on coding rather than tooling [26].

Fig. 1 illustrates the interaction between a language client and a language server through the LSP. Communication begins with an initialization request, which establishes the capabilities of both sides and prepares the environment for subsequent tasks. As the developer opens or edits files, notifications such as didOpen and didChange ensure synchronization between client and server, despite running as separate processes. This design enables the server to deliver real-time diagnostics and respond to user requests, such as retrieving symbol definitions, thereby enhancing the efficiency of the development workflow. Once a document is closed, the server discards its in-memory state while retaining the saved version, preventing unnecessary resource consumption.

The application of LSP concepts to security tools is increasingly evident in systems such as SonarQube and Semgrep. Extending this approach with ML creates further opportunities: a language server can analyze code as it is written, identifying potential vulnerabilities with greater immediacy than conventional static analysis tools. However, most current solutions still lack fully integrated real-time detection within development environments, especially at a fine-grained, line-level resolution. Additionally, there is limited research on combining lightweight ML models with LSP to provide rapid feedback without disrupting developer workflows. This gap highlights the need for solutions that not only detect vulnerabilities accurately but also seamlessly integrate into live coding processes, offering developers actionable insights immediately as they write code.

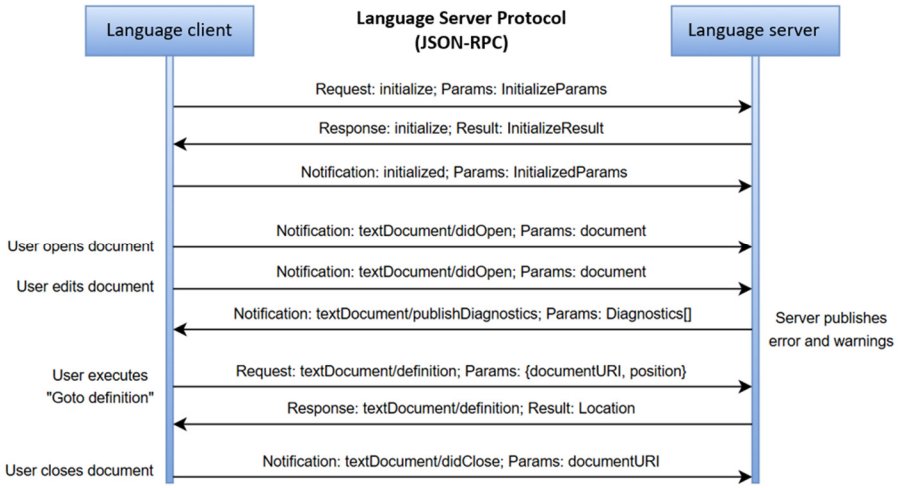


Fig. 1 LSP-based communication during a routine editing session

3. System Architecture

The proposed system architecture, as illustrated in Fig. 2, comprises three primary components: the language client, the language server, and the analysis component. Together, these modules make real-time vulnerability detection possible during code development. The language client and server interact using the LSP, which manages communication between the two processes. Within the server, the analysis component performs the core task of detecting and classifying vulnerabilities, thereby linking the communication framework with the system's security function.

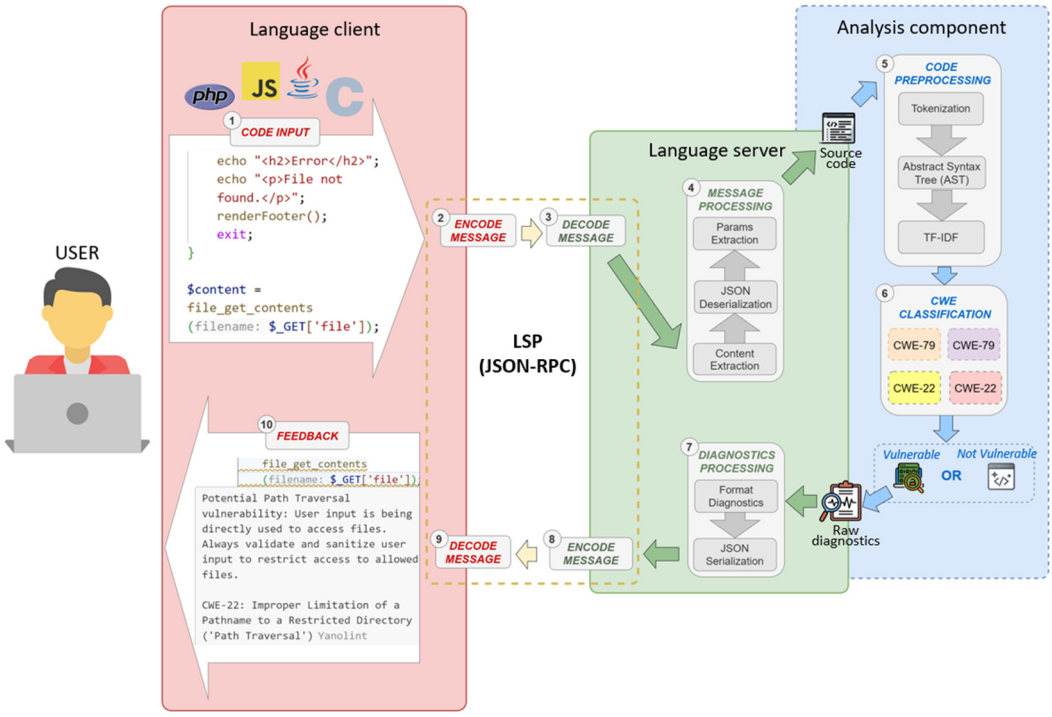


Fig. 2 Proposed system architecture

3.1. Language client

The language client is the user-facing component that facilitates code input and feedback delivery. Users interact with this component through their integrated development environment (IDE), where they write code and initiate vulnerability detection processes. When user actions, such as file creation or modification, are detected, the client initiates the message encoding step. This process constructs a message in compliance with the JSON-RPC specifications of the LSP. The encoded message includes a UTF-8-formatted header and body that detail the invoked method and parameters, such as the source code. For instance, a typical message specifies the content length, JSON-RPC version, and relevant metadata required for further processing. Once the message has been constructed, it is transmitted to the language server for analysis and processing.

After receiving diagnostic results from the language server, the client initiates the message decoding step to process and present actionable insights from the server's response. The data is translated into a user-friendly format and displayed directly within the IDE. The feedback highlights vulnerabilities in the code and offers mitigation recommendations, creating a secure and developer-friendly environment.

3.2. Language server

The language server acts as the intermediary between the language client and the analysis component, interpreting messages from the client and managing the diagnostic pipeline. Upon receiving a message from the client, the language server begins the message decoding process. This is followed by message processing, which involves three sub-stages: content extraction, JSON deserialization, and params extraction. Content extraction retrieves the JSON content, which includes critical metadata. JSON deserialization converts the JSON into a structured format, enabling parameter extraction to isolate essential parameters, such as the source code, for further analysis by the analysis component. The processed data is then forwarded for feature extraction and classification.

Once the analysis component returns diagnostic results, the language server initiates the diagnostics processing phase to prepare feedback for the client. This involves compiling and formatting the analysis results into structured diagnostic reports. The findings are then serialized into JSON, ensuring the content conforms to LSP specifications. Finally, the formatted diagnostics are encoded into a JSON-RPC-compliant message and sent back to the language client, completing the diagnostic cycle and enabling real-time feedback.

3.3. Analysis component

The analysis component operates as the core of the detection pipeline, transforming raw source code into structured insights on security weaknesses. The process begins with code preprocessing, where tokenization, abstract syntax tree (AST) traversal, and TF-IDF vectorization convert the source code into a numerical form suitable for learning-based evaluation. These representations are then analyzed in the classification stage, where a trained model determines the presence of vulnerabilities. Detected weaknesses are immediately mapped to their corresponding CWE identifiers, such as CWE-22 (Path Traversal) or CWE-434 (Unrestricted File Upload), thereby linking the output to widely recognized security taxonomies. This mapping ensures that the feedback is not only a binary assessment of vulnerability or non-vulnerability, but also an explanation of the flaw type and its implications for software security. The results are returned to the language server for integration into developer feedback, completing a cycle that embeds security diagnostics directly into the coding workflow.

4. Results and Discussion

To validate the proposed architecture, a proof-of-concept prototype was implemented and tested on a Windows 11 (64-bit) system using Windows Subsystem for Linux (WSL), powered by an Intel Core i7 (11th Gen) CPU and 24 GB of RAM. This prototype demonstrates the system's practical relevance and establishes a baseline for future scalability and deployment.

The language client, built with Neovim and Lua, interacts with the language server via the LSP and supports PHP, JavaScript, Java, and C files for real-time analysis. Error-handling routines ensure smooth operation by notifying users of initialization issues. The language server, developed in Go and compiled into a portable binary, efficiently processes JSON-RPC requests and extracts source code and metadata for analysis and interpretation.

The analysis component, implemented in Python using the scikit-learn library, utilized an ML model trained and evaluated on two datasets: CVEfixes [27] and MoreFixes [28]. CVEfixes consists of vulnerability-related commits that explicitly reference CVE identifiers, contain CVE IDs, descriptions, CWE classifications, and corresponding code changes. MoreFixes extends this by including a broader collection of vulnerability patches, not tied to CVE IDs, but validated as true security fixes, thereby enhancing dataset diversity and representativeness. Both datasets were randomly split into 80% training, 10% validation, and 10% testing, following the approach in Bhandari et al. [27]. Approximately 20,000 labeled code lines were used, with about 40% marked as vulnerable, ensuring a balanced evaluation setting.

Model evaluation employed accuracy, precision, recall, F1-score, and the Area Under the Precision–Recall Curve (PR-AUC) as performance metrics. At the same time, the language client and language server were tested with 50 vulnerable samples from the testing set to measure processing time and diagnostic feedback accuracy.

4.1. Language client simulation result

The language client in this study was subjected to the following testing and evaluation requirements. It should be able to capture code inputs or changes from a specific file, notify the user of vulnerabilities detected within the file by highlighting the corresponding lines of code, and provide a detailed description and CWE classification when the highlighted line is hovered over. Testing involved using sample source files containing vulnerabilities to evaluate the functionality and accuracy of the language client.

In testing the language client, 50 source codes with vulnerable lines from the testing set were copied into a file to trigger the vulnerability detection process. These samples were sourced from a published dataset called CVEfixes [27], a comprehensive collection of vulnerability data that includes CVE-IDs, descriptions, CWE classifications, and code changes, facilitating automated vulnerability prediction and classification. Evaluation of the outlined conditions revealed vulnerabilities as they were flagged in real-time. Results showed that the language client successfully detected vulnerabilities within the code and displayed appropriate notifications. Vulnerable lines were visually indicated using an in-line diagnostic message specifying the nature of the issue (Fig. 3).

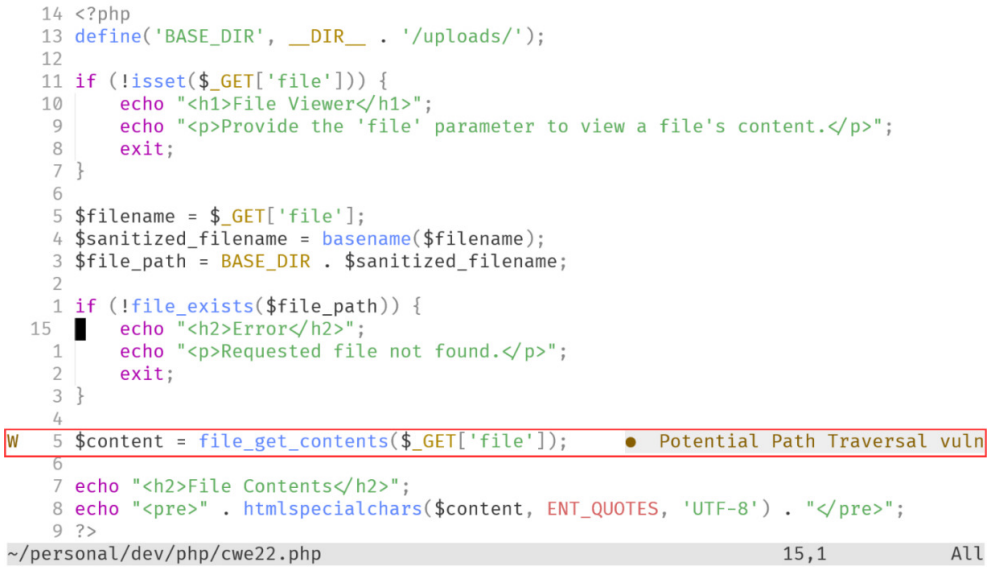


Fig. 3 Real-time diagnostic notification of highlighted vulnerable lines

The language client played a vital role in enhancing the developer's awareness of security vulnerabilities within the IDE. Whenever a vulnerability was detected, a visual marker notified the user. Additional details about vulnerability, including a CWE classification and description, were displayed when the user hovered over the highlighted line (Fig. 4). This functionality ensured that developers received actionable insights into the vulnerabilities present in their code, thereby facilitating a secure development workflow. The results demonstrate the effectiveness of the language client in achieving its intended purpose of real-time vulnerability detection and feedback delivery.

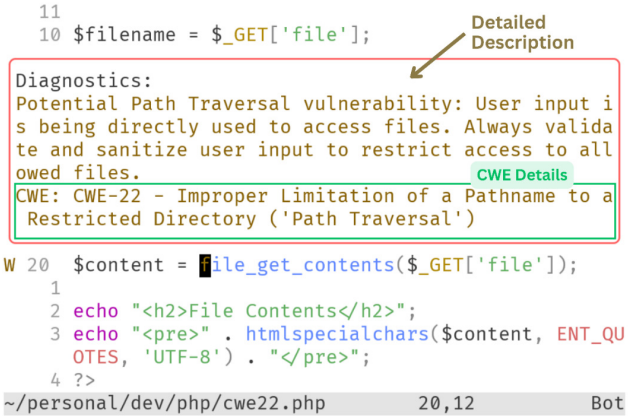


Fig. 4 CWE classification and vulnerability details in the code editor

4.2. Language server performance evaluation

As illustrated in the proposed system architecture (Fig. 2), the language server plays a pivotal role in decoding messages from the language client and managing the diagnostic pipeline. The process begins with the language server receiving an encoded message from the language client, which includes the source code and relevant metadata. The message decoding step involves extracting the JSON content, followed by JSON deserialization to convert the data into a structured format. This allows for parameter extraction, where essential parameters such as the source code are isolated for further analysis. The overall time recorded for these initial steps was 0.068 milliseconds, ensuring efficient handling of incoming requests.

Following the initial decoding, the language server proceeds to the diagnostics processing phase, where the analysis results from the analysis component are compiled and formatted into structured diagnostic reports. This phase involves formatting the diagnostics, serializing them into JSON, and encoding the final message in compliance with the LSP, as illustrated in Fig. 5. The time required for this phase was 0.032 milliseconds, demonstrating the system's ability to prepare and transmit feedback efficiently. The entire process, from receiving the message to sending the RPC message with diagnostics back to the language client, was completed in approximately 72 milliseconds, demonstrating the system's responsiveness and efficiency.

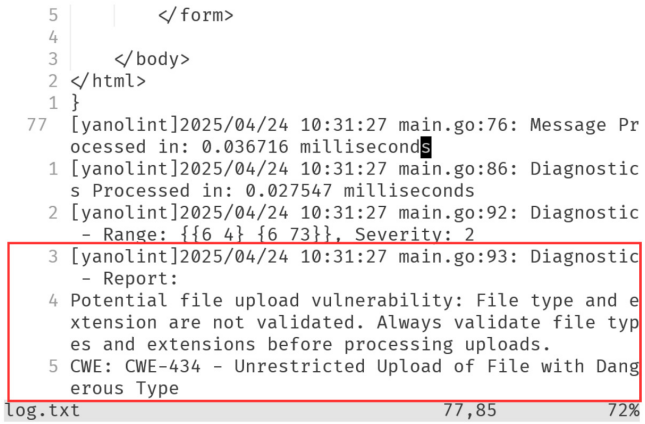


Fig. 5 Diagnostic processing and message exchange log

The evaluation was conducted to validate this measurement using 50 vulnerable code samples drawn from the chosen dataset, each containing roughly 50 to 100 lines of source code. It is essential to note that several factors can impact this metric, including the size and complexity of the input code, the efficiency of feature extraction in the analysis component, hardware specifications, and concurrent system workload. The simulation results were logged into a text file to ensure accuracy and traceability (Fig. 5). The logged data confirmed that the language server effectively managed the diagnostic pipeline, providing real-time feedback to the language client. This efficiency is crucial for maintaining a secure and developer-friendly environment, as it enables the immediate identification and mitigation of potential vulnerabilities in the code. The results underscore the robustness of the language server in handling complex diagnostic tasks within a minimal timeframe, ensuring a seamless integration with the overall system architecture.

4.3 Performance evaluation of the analysis component

The analysis component employs an ML approach to classify source code lines as either vulnerable or safe, using a Random Forest classifier implemented in Python with the scikit-learn library. The model was trained on the CVEfixes [27] and MoreFixes [28] datasets, both of which contain real-world vulnerability fixes drawn from open-source repositories. These datasets provide paired versions of code before and after remediation, enabling fine-grained identification of vulnerable and non-vulnerable segments. Only PHP, JavaScript, Java, and C files were retained to align with the supported languages of the language client.

For labeling, lines of code present in the insecure version but modified or removed during the fix were labeled as vulnerable (buggy). In contrast, the newly added or replaced lines in the corrected version were labeled as non-vulnerable (fixed). The labeling was validated through repository metadata and commit messages to ensure that each change represented a genuine security fix. To address potential class imbalance, samples were stratified to maintain an approximate 40:60 ratio of vulnerable to non-vulnerable lines. This balanced composition enabled the model to capture patterns from both classes effectively, preventing any inclination toward non-vulnerable code.

For preprocessing, comments, blank lines, and unnecessary whitespace were removed to reduce noise in the training data. Each code line was tokenized into lexical elements such as keywords, operators, and identifiers to capture syntactic information. AST traversal was then applied to extract structural relationships between code components, including control flows and function dependencies. Finally, TF-IDF vectorization converted the tokens and AST-derived features into numerical vectors that quantify the relative importance of code elements for model training.

To ensure robustness and improve generalization, the dataset was divided into three distinct subsets: 80% for training, 10% for validation, and 10% for testing, in alignment with the methodology described in Bhandari et al. [27]. The validation portion was employed to optimize hyperparameters—such as the depth and quantity of trees in the ensemble model—and to refine feature selection strategies for improved predictive accuracy. The dataset was deliberately centered on critical vulnerability types, namely CWE-79 (Cross-Site Scripting), CWE-89 (SQL Injection), CWE-22 (Path Traversal), and CWE-434 (Unrestricted File Upload). These categories, recognized within the CWE Top 25 Most Dangerous Software Weaknesses and commonly found in real-world projects [29], ensured that the model focused on vulnerabilities with meaningful real-world impact. Code samples were sourced primarily from PHP, JavaScript, Java, and C, corresponding to the languages supported by the language client.

Model evaluation employed standard performance indicators: accuracy, precision, recall, and the F1-score. Accuracy represents the fraction of correct predictions relative to all samples. Precision measures the proportion of true positives among all identified positives, indicating how reliably the model flags vulnerabilities. Recall quantifies how many actual vulnerable instances were correctly detected. Finally, the F1-score harmonizes precision and recall into a single, balanced measure, providing a comprehensive view of classification performance.

$$\text{Accuracy} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{TN} + \text{FP} + \text{FN}} \tag{1}$$

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}} \tag{2}$$

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}} \tag{3}$$

$$\text{F1-score} = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} \tag{4}$$

where TP, TN, FP, and FN denote true positives, true negatives, false positives, and false negatives, respectively.

To provide a more robust assessment that compensates for the dataset's slight class imbalance, the PR-AUC was employed as an additional evaluation metric. This measure is especially valuable for imbalanced data, as it captures model behavior across varying decision thresholds. A higher PR-AUC value reflects a stronger capability in detecting vulnerabilities while simultaneously reducing false positive occurrences.

Table 1 presents the model's evaluation results on line-level samples derived from the CVEfixes [27] and MoreFixes [28] datasets, with bolded values denoting the highest performance achieved for each metric. On the CVEfixes dataset, the model attained an accuracy of 82.3%, indicating a strong capability to distinguish between secure and insecure lines of code. A precision value of 81.8% reflects effective minimization of false positives, while a recall rate of 79.6% demonstrates consistent identification of genuine vulnerabilities. The resulting F1-score of 80.7% signifies a well-balanced integration of precision and recall, underscoring the model's overall reliability in detecting security threats. Moreover, the PR-AUC of 90.1% emphasizes the model's robust discriminative power in classifying vulnerable versus non-vulnerable code lines.

Table 1 Performance metrics

Metric	Analysis component (with CVEfixes)	Analysis component (with MoreFixes)
Accuracy (%)	**82.3**	82.0
Precision (%)	**81.8**	75.7
Recall (%)	79.6	**81.1**
F1-score (%)	**80.7**	78.3
PR-AUC (%)	**90.1**	87.98

Note: Bold values indicate the best performance

Evaluation on the MoreFixes dataset showed that the model delivered comparable performance, recording an accuracy of 82.0% and an F1-score of 78.3%. While precision experienced a slight drop to 75.7%, an increase in recall to 81.1% indicated that the model became more proactive in labeling lines as potentially vulnerable within the larger dataset. The PR-AUC value of 87.98% further confirmed the model's stable capacity to differentiate between vulnerable and non-vulnerable code segments. Overall, these outcomes highlight the system's robustness across datasets of differing sizes and validate the consistency of its ML framework. The experiments were conducted at a line-level granularity, treating each line of code as an independent classification sample. The dataset itself contained roughly 20,000 labeled code lines, about 40% of which were identified as vulnerable, providing a balanced and realistic evaluation setting for vulnerability detection.

Fig. 6 illustrates the confusion matrix for the CVEfixes dataset. Out of 20,000 line-level samples, the model correctly classified 10,583 non-vulnerable lines and 6,368 vulnerable lines. It also produced 1,417 false positives, where benign lines were mistakenly labeled as vulnerable, and 1,632 false negatives, representing undetected vulnerabilities. The comparable proportion of these misclassifications indicates the effectiveness of the labeling approach, in which patched lines were assigned as non-vulnerable and buggy segments were assigned as vulnerable. Overall, these findings demonstrate that the model consistently and reliably detects line-level vulnerabilities within CVE-referenced patches.

Fig. 7 presents the confusion matrix for the MoreFixes dataset. The model correctly identified 9,900 non-vulnerable and 6,500 vulnerable lines, producing 2,088 false positives and 1,512 false negatives. Compared with CVEfixes, MoreFixes shows a lower precision but higher recall, indicating that the model flagged more potential vulnerabilities at the expense of additional false alarms. This reflects the greater diversity of MoreFixes, which incorporates non-CVE-linked fixes and broader code contexts. Despite this, the model generalizes effectively, maintaining robust detection performance across datasets beyond CVE-referenced vulnerabilities.

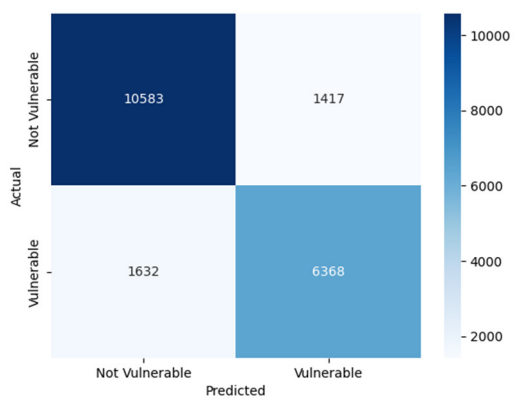


Fig. 6 Confusion matrix for the CVEfixes dataset

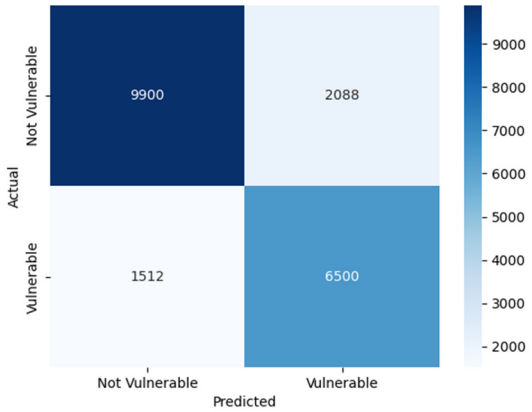


Fig. 7 Confusion matrix for the MoreFixes dataset

Fig. 8 illustrates the PR curves comparing model performance on the CVEfixes and MoreFixes datasets. The CVEfixes curve (PR-AUC = 0.9010) slightly surpasses MoreFixes (PR-AUC = 0.8798), showing stronger PR balance in detecting CVE-linked vulnerabilities. Both curves remain well above the 0.40 baseline, confirming reliable identification of vulnerable lines despite the mild class imbalance. The lower PR-AUC of MoreFixes reflects its broader and more varied code contexts, which increases the difficulty of detection. Overall, the model demonstrates consistent generalization across datasets, maintaining dependable vulnerability detection performance.

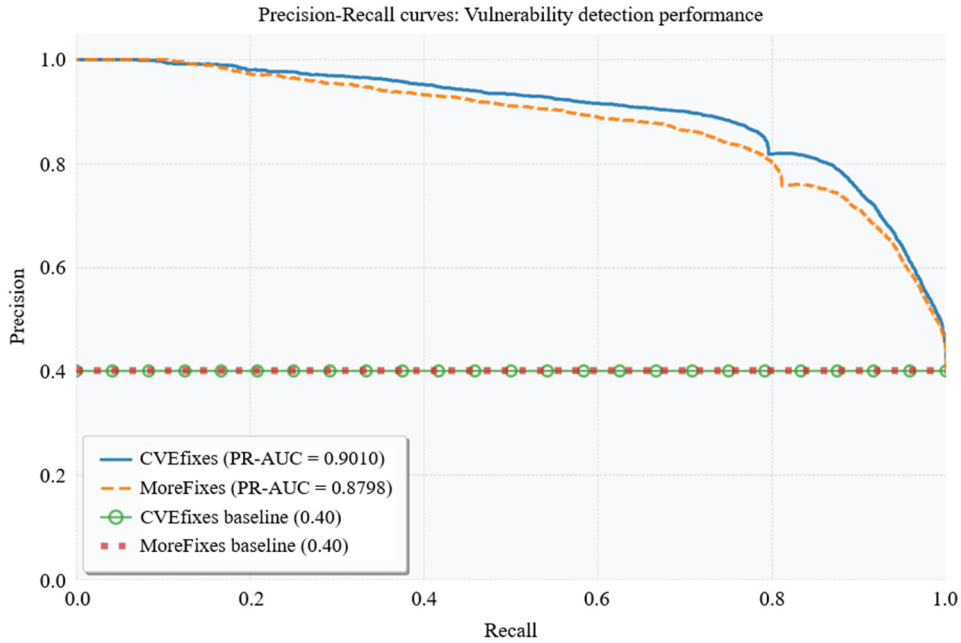


Fig. 8 PR curves comparing model performance on CVEfixes and MoreFixes datasets

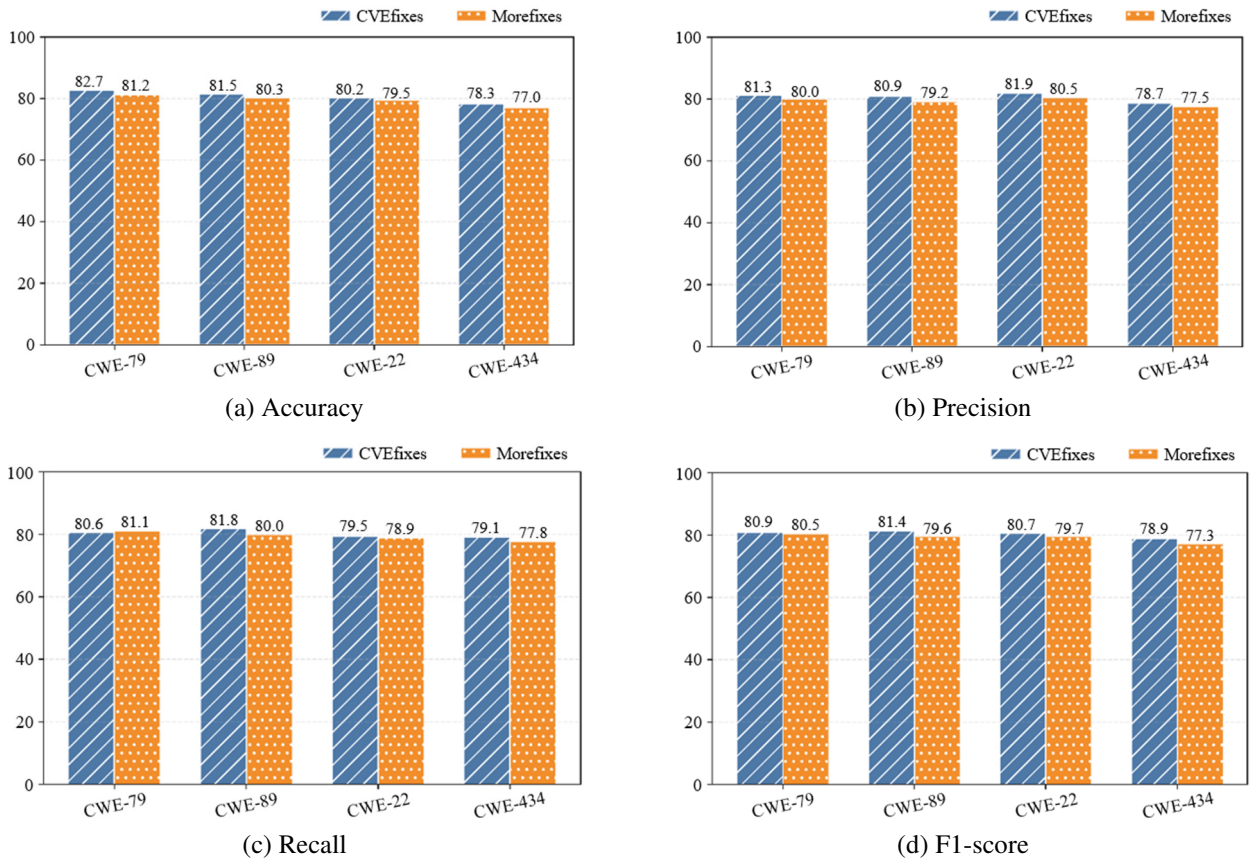
Table 2 presents the model's detection performance across CWE categories, with bold values denoting the best results for each metric. Fig. 9 illustrates trends across the four CWE types. In the CVEfixes dataset, CWE-79 (Cross-Site Scripting) achieved the highest accuracy (82.7%) and F1-score (80.9%), followed by CWE-89 (SQL Injection) with 81.5% accuracy and an F1-score of 81.4%. CWE-22 (Path Traversal) reached the highest precision (81.9%), while CWE-434 (Unrestricted File

Upload) had the lowest F1-score (78.9%). These results indicate that TF-IDF and basic AST features are insufficient to accurately encode complex semantics, such as file validation and permission checks. Although Random Forest captures syntactic patterns effectively, deeper models like Transformers or GNNs are better suited for reasoning over multi-step dependencies. The highest PR-AUC (74.0%) confirms strong discriminative capability, particularly for CWE-79.

Table 2 CWE-specific detection performance

	CWE Identifier	CWE-79	CWE-89	CWE-22	CWE-434
	Description	Cross-Site Scripting	SQL Injection	Path Traversal	Unrestricted Upload of Dangerous Files
CVEfixes	Accuracy (%)	**82.7**	81.5	80.2	78.3
	Precision (%)	81.3	80.9	**81.9**	78.7
	Recall (%)	80.6	**81.8**	79.5	79.1
	F1-score (%)	80.9	**81.4**	80.7	78.9
	PR-AUC (%)	**74.0**	72.0	71.0	69.0
MoreFixes	Accuracy (%)	**81.2**	80.3	79.5	77.0
	Precision (%)	80.2	79.2	**80.5**	77.5
	Recall (%)	**81.1**	80.0	78.9	77.8
	F1-score (%)	**80.5**	79.6	79.7	77.9
	PR-AUC (%)	**69.0**	67.0	66.0	64.0

Note: Bold values indicate the best performance



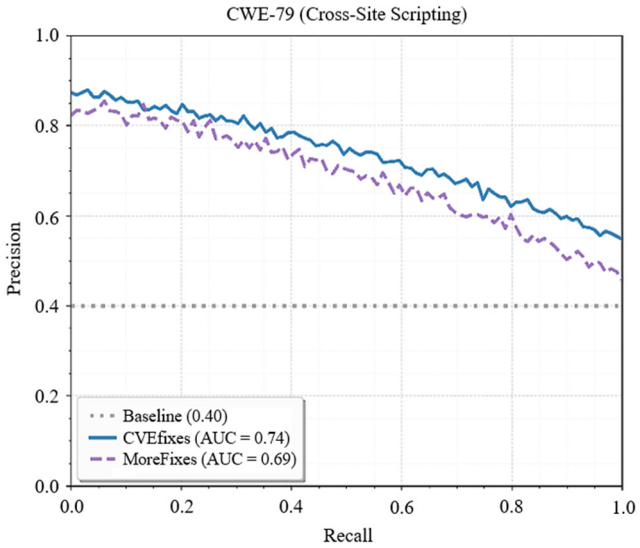
(a) Accuracy (b) Precision

(c) Recall (d) F1-score

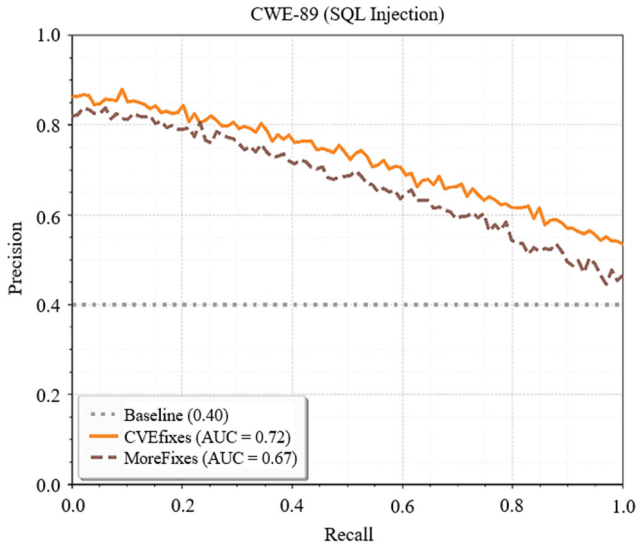
Fig. 9 Detection results by CWE type

In the MoreFixes dataset, performance declined slightly due to greater code variability. CWE-79 and CWE-89 remained top performers, with F1-scores of 80.5% and 79.6%, indicating stable detection of input-related vulnerabilities. CWE-22 achieved high precision (80.5%) but moderate recall, while CWE-434 remained the weakest with 77.0% accuracy and 77.9% F1-score. These outcomes suggest that future work should incorporate semantic embeddings or graph-based models to capture contextual and logic-dependent flaws. Despite the increased complexity of the dataset, the top PR-AUC (69.0%) still demonstrates a reliable distinction between secure and vulnerable code lines.

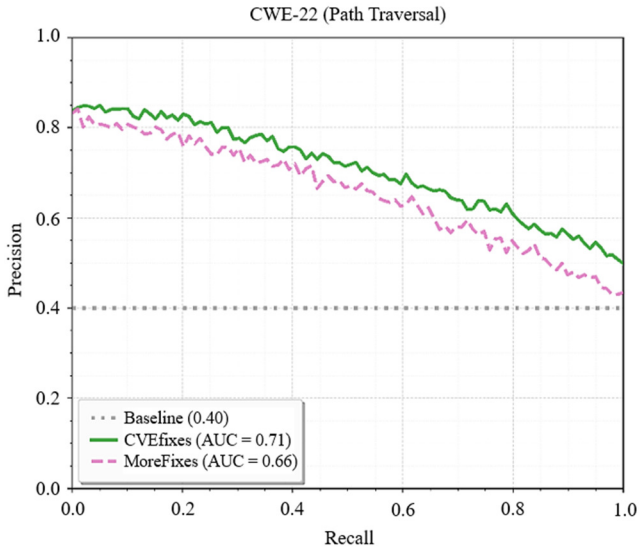
Fig. 10 illustrates the PR curves for the four CWE categories across both the CVEfixes and MoreFixes datasets. Fig. 10(a) shows that CWE-79 (Cross-Site Scripting) achieves the highest area under the curve (AUC) of 0.74 in CVEfixes, reflecting strong and stable precision across all recall levels. A similar trend is observed in Fig. 10(b) for CWE-89 (SQL Injection), which attains an AUC of 0.72, indicating reliable recognition of SQLi patterns. Fig. 10(c) presents the results for CWE-22 (Path Traversal), which yields an AUC of 0.71, suggesting moderate stability in precision as recall increases. In contrast, Fig. 10(d) reveals that CWE-434 (Unrestricted File Upload) obtains the lowest AUC values (0.69 for CVEfixes and 0.64 for MoreFixes), highlighting its weaker generalization capability and higher misclassification rate. Collectively, these results demonstrate that the models perform best on Cross-Site Scripting and SQL Injection, while Unrestricted File Upload remains the most challenging category to detect accurately.



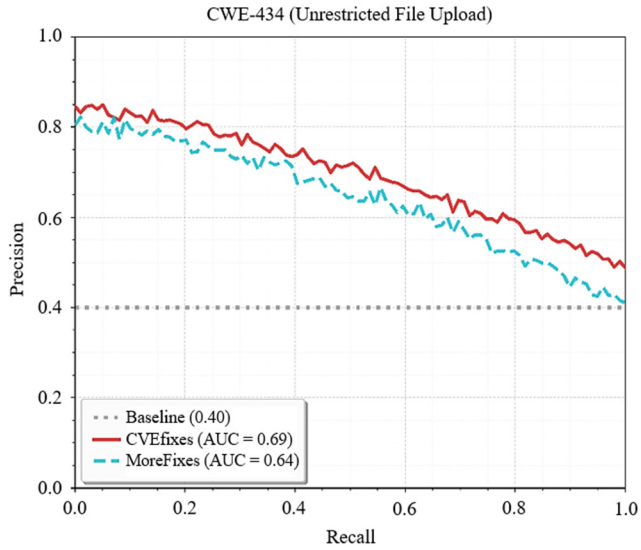
(a) PR curve for CWE-79



(b) PR curve for CWE-89



(c) PR curve for CWE-22



(d) PR curve for CWE-434

Fig. 10 PR curves by CWE category on the CVEfixes and MoreFixes datasets

5. Conclusion

This study presents a real-time line-level vulnerability detection system that integrates an ML-based analysis component into a language server framework to enhance software security during development. The system processes source code through tokenization, AST generation, and TF-IDF vectorization, with a Random Forest classifier used to identify vulnerabilities at the line level. The model was trained and tested on the CVEfixes and MoreFixes datasets, covering vulnerabilities such as CWE-

79 (Cross-Site Scripting), CWE-89 (SQL Injection), CWE-22 (Path Traversal), and CWE-434 (Unrestricted File Upload). Experimental results demonstrate consistent performance across datasets, achieving F1-scores above 80% for major categories, validating the feasibility of lightweight ML-based real-time detection. The key contributions of this study are as follows:

(1) Real-time line-level detection: The system introduces a proactive approach by embedding line-level ML-driven vulnerability detection within the development process, providing immediate and fine-grained feedback as code is written, with coverage across multiple CWE categories.

(2) Efficient lightweight architecture: By combining TF-IDF and AST-based feature extraction with a Random Forest model, the approach achieves high accuracy while maintaining minimal computational overhead, ensuring real-time responsiveness.

(3) Seamless workflow integration: Implemented through the LSP, the system demonstrates how ML-based security analysis can be integrated into modern IDEs to enhance secure coding practices.

Despite strong results, the system's limitations arise from its ML framework. The weaker performance on CWE-434 indicates that the Random Forest model, combined with TF-IDF and AST features, struggles with vulnerabilities governed by complex semantics rather than clear syntactic cues. This reflects a trade-off that favors a lightweight, high-speed design suitable for real-time feedback over more computationally demanding deep learning methods. Future work should address this gap through hybrid or graph-based architectures (e.g., Graph Neural Networks, GNNs) and contextual embeddings (e.g., CodeBERT), which can model both semantic and data-flow relationships. Such architectures are likely to enhance the detection of logic-dependent vulnerabilities, including those related to Unrestricted File Uploads. Broader evaluation across multiple programming languages, software domains, and development environments will further validate scalability and robustness.

Conflicts of Interest

The authors declare no conflict of interest.

References

[1] J. S. Cabrera, A. R. L. Reyes, and C. A. Lasco, "Multicriteria Decision Analysis on Information Security Policy: A Prioritization Approach," Advances in Technology Innovation, vol. 6, no. 1, pp. 31-38, 2021.

[2] F. Spanca and A. Salihu, "Unveiling the Consequences of Data Breaches: Risks, Impacts, and Mitigation in the Digital Age," International Conference on Electrical, Communication and Computer Engineering, pp. 1-8, 2024.

[3] M. Alenezi and M. Zarour, "On the Relationship between Software Complexity and Security," https://doi.org/10.48550/arXiv.2002.07135, 2020.

[4] C. R. Jose, "Exploring Security Process Improvements for Integrating Security Tools within a Software Application Development Methodology," Ph.D. dissertation, Colorado Technical University, Colorado, CO, 2020.

[5] Ö. Aslan, S. S. Aktuğ, M. Ozkan-Okay, A. A. Yilmaz, and E. Akin, "A Comprehensive Review of Cyber Security Vulnerabilities, Threats, Attacks, and Solutions," Electronics, vol. 12, no. 6, article no. 1333, 2023.

[6] H. Hanif, M. H. N. Md Nasir, M. F. Ab Razak, A. Firdaus, and N. B. Anuar, "The Rise of Software Vulnerability: Taxonomy of Software Vulnerabilities Detection and Machine Learning Approaches," Journal of Network and Computer Applications, vol. 179, article no. 103009, 2021.

[7] S. Pargaonkar, "Advancements in Security Testing: A Comprehensive Review of Methodologies and Emerging Trends in Software Quality Engineering," International Journal of Science and Research, vol. 12, no. 9, pp. 61-66, 2023.

[8] "CVE-2024-9465," https://www.cve.org/CVERecord?id=CVE-2024-9465, accessed in 2025.

[9] "CVE-2024-51378," https://www.cve.org/CVERecord?id=CVE-2024-51378, accessed in 2025.

[10] K. A. Deepak, C. Gnanaprakasam, S. N. Prabu, N. Senthamilarasi, K. J. Chenni, R. R. Vinston, et al., "Vulnerability Detection in Software Applications Using Static Code Analysis," Journal of Theoretical and Applied Information Technology, vol. 102, no. 4, pp. 1307-1320, 2024.

[11] M. Fu and C. Tantithamthavorn, "LineVul: A Transformer-Based Line-Level Vulnerability Prediction," Proceedings of the 19th International Conference on Mining Software Repositories, pp. 608-620, 2022.

[12] D. Hin, A. Kan, H. Chen, and M. A. Babar, "LineVD: Statement-Level Vulnerability Detection Using Graph Neural Networks," Proceedings of the 19th International Conference on Mining Software Repositories, pp. 596-607, 2022.
[13] Y. Wu, R. A. Gandhi, and H. Siy, "Using Semantic Templates to Study Vulnerabilities Recorded in Large Software Repositories," Proceedings of the 2010 ICSE Workshop on Software Engineering for Secure Systems, pp. 22-28, 2010.
[14] "CWE-79: Improper Neutralization of Input during Web Page Generation ('Cross-Site Scripting')," https://cwe.mitre.org/data/definitions/79.html, accessed in 2025.
[15] M. Liu, B. Zhang, W. Chen, and X. Zhang, "A Survey of Exploitation and Detection Methods of XSS Vulnerabilities," IEEE Access, vol. 7, pp. 182004-182016, 2019.
[16] A. R. L. Reyes, E. D. Festijo, and R. P. Medina, "Enhanced Multi-Factor Out-of-Band Authentication En Route to Securing SMS-Based OTP," International Journal of Engineering and Technology Innovation, vol. 9, no. 2, pp. 145-154, 2019.
[17] A. R. L. Reyes, E. D. Festijo, and R. P. Medina, "Securing One Time Password (OTP) for Multi-Factor Out-of-Band Authentication through a 128-bit Blowfish Algorithm," International Journal of Communication Networks and Information Security, vol. 10, no. 1, pp. 242-247, 2018.
[18] W. Charoenwet, P. Thongtanunam, V. T. Pham, and C. Treude, "Toward Effective Secure Code Reviews: An Empirical Study of Security-Related Coding Weaknesses," Empirical Software Engineering, vol. 29, no. 4, article no. 88, 2024.
[19] Z. Li, Z. Liu, W. K. Wong, P. Ma, and S. Wang, "Evaluating C/C++ Vulnerability Detectability of Query-Based Static Application Security Testing Tools," IEEE Transactions on Dependable and Secure Computing, vol. 21, no. 5, pp. 4600-4618, 2024.
[20] T. Marjanov, I. Pashchenko, and F. Massacci, "Machine Learning for Source Code Vulnerability Detection: What Works and What Isn't There Yet," IEEE Security & Privacy, vol. 20, no. 5, pp. 60-76, 2022.
[21] P. Dubey, P. Dubey, and P. N. Bokoro, "Unpacking Sarcasm: A Contextual and Transformer-Based Approach for Improved Detection," Computers, vol. 14, no. 3, article no. 95, 2025.
[22] S. Rajapaksha, J. Senanayake, H. Kalutarage, and M. O. Al-Kadri, "AI-Powered Vulnerability Detection for Secure Source Code Development," International Conference on Information Technology and Communications Security, vol. 13809, pp. 275-288, 2022.
[23] Y. Yang and H. Wang, "Random Forest-Based Machine Failure Prediction: A Performance Comparison," Applied Sciences, vol. 15, no. 16, article no. 8841, 2025.
[24] S. Zaharia, T. Rebedea, and S. Trausan-Matu, "Machine Learning-Based Security Pattern Recognition Techniques for Code Developers," Applied Sciences, vol. 12, no. 23, article no. 12463, 2022.
[25] "Official page for Language Server Protocol," https://microsoft.github.io/language-server-protocol/, accessed in 2025.
[26] D. Bork and P. Langer, "Language Server Protocol: An Introduction to the Protocol, Its Use, and Adoption for Web Modeling Tools," Enterprise Modelling and Information Systems Architectures, vol. 18, pp. 9:1-16, 2023.
[27] G. Bhandari, A. Naseer, and L. Moonen, "CVEfixes: Automated Collection of Vulnerabilities and Their Fixes from Open-Source Software," Proceedings of the 17th International Conference on Predictive Models and Data Analytics in Software Engineering, pp. 30-39, 2021.
[28] J. Akhoundali, S. R. Nouri, K. Rietveld, and O. Gadyatskaya, "MoreFixes: A Large-Scale Dataset of CVE Fix Commits Mined through Enhanced Repository Discovery," Proceedings of the 20th International Conference on Predictive Models and Data Analytics in Software Engineering, pp. 42-51, 2024.
[29] H. Pearce, B. Ahmad, B. Tan, B. Dolan-Gavitt, and R. Karri, "Asleep at the Keyboard? Assessing the Security of GitHub Copilot's Code Contributions," Communications of the ACM, vol. 68, no. 2, pp. 96-105, 2025.