CLASSIFICATION OF SOURCE CODE VULNERABILITIES AND ANALYSIS OF DETECTION METHODS: EVALUATION, COMPARISON, AND PROPOSED APPROACHES

Nin Ho Le Viet[1,2], Chieu Ta Quang[2], Cuong Dang Van[3]and Tuan Nguyen Kim[3]

[1]School of Computer Science, Duy Tan University, Danang, Vietnam
[2]Faculty of Computer Science and Engineering, Thuyloi University, Hanoi, Vietnam
[3]Phenikaa School of Computing, Phenikaa University, Hanoi, Vietnam

ABSTRACT

Source code vulnerabilities are an important cause leading to many information security incidents in modern software systems. Due to the diversity of causes and technical characteristics, source code vulnerability detection is difficult to be effectively addressed by a single method. This paper focuses on classifying, evaluating, and comparing source code vulnerability detection methods in order to provide a systematic view of this problem. The first contribution of the paper is to develop a classification approach for source code vulnerabilities based on causes and technical characteristics, thereby clarifying the nature of each vulnerability group. Next, the paper analyzes and evaluates common detection method categories, including static analysis, dynamic analysis, and machine learning and deep learning based methods, with consideration of the application scope and characteristic limitations of each approach. On that basis, the paper conducts an overall comparison among the methods to indicate that there does not exist a single technique that can effectively detect every type of source code vulnerability. Finally, the paper proposes a combined approach to leverage the advantages of different methods in source code vulnerability detection. The results and analyses in the paper are expected to support researchers and software engineers in evaluating, selecting, and orienting the development of vulnerability detection solutions in the future.

KEYWORDS

Source code vulnerabilities, Software security, Vulnerability detection, Static and dynamic analysis, Machine learning

1. INTRODUCTION

In modern information systems, software plays a central role in delivering services across critical domains such as e-government, finance, healthcare, cloud computing, and artificial intelligence. As software systems become larger and more interconnected, source code vulnerabilities have remained a major root cause of security incidents, leading to system compromise, data leakage, and service disruption. In practice, vulnerabilities may arise from diverse causes and manifest in different technical forms, ranging from memory management errors such as buffer overflow [1], out-of-bounds access [2], and use-after-free [3], to input handling flaws such as SQL injection [4], command injection [5], and format string vulnerabilities [6]. Many vulnerabilities, especially those depending on program logic and context, may not trigger immediate runtime failures and only appear under specific execution scenarios, which makes detection particularly difficult.

To address source code vulnerabilities, researchers and practitioners have developed multiple

DOI: 10.5121/ijnsa.2026.18201

detection approaches. Static analysis inspects programs without execution, leveraging syntactic rules and program analysis techniques to identify suspicious patterns early in the development process. While widely adopted, static analysis often suffers from high false positives and limited capability in handling context-dependent vulnerabilities. Dynamic analysis and fuzzing, in contrast, focus on runtime behaviors and input-driven testing to uncover vulnerabilities that only manifest during execution. Although such techniques can provide stronger evidence of exploitability, their effectiveness depends heavily on test coverage and the diversity of generated inputs. More recently, machine learning and deep learning based methods have been proposed to exploit statistical and structural features of source code for automated vulnerability detection [7]. These learning-based approaches reduce reliance on handcrafted rules, yet their performance still depends on the quality of training data, feature representation, and generalization capability.

Despite notable progress, source code vulnerability detection remains a complex problem in which no single technique can reliably detect all vulnerability types across real-world settings. Many existing works focus on improving individual detection methods or reporting performance on limited vulnerability sets, while the relationship between vulnerability characteristics and method suitability is often not made explicit. As a result, selecting appropriate detection techniques for complex, multi-language software systems remains challenging. This observation motivates the need for a systematic perspective that (i) Clarifies vulnerability categories, (ii) Analyzes detection methods under consistent criteria, and (iii) Supports hybrid-oriented thinking, where complementary methods can be combined to mitigate individual limitations.

Motivation and Scope: The motivation of this study arises from the observation that source code vulnerability detection research is often fragmented across different methodological directions. While numerous works propose improvements for individual techniques, fewer studies provide a structured comparison that explicitly links vulnerability characteristics to detection capabilities. As a result, method selection in practice is frequently driven by tool availability rather than systematic understanding. The scope of this paper is therefore analytical rather than algorithm-centric. The study does not aim to introduce a new state-of-the-art detection model, nor to conduct large-scale empirical benchmarking. Instead, it focuses on organizing vulnerability categories, examining major detection paradigms under consistent evaluation criteria, and discussing a hybrid-oriented direction supported by illustrative validation. By clarifying relationships among vulnerability types and detection strategies, the paper seeks to provide a conceptual and comparative framework that supports informed decision making in both research and practical deployment.

To address this need, this paper adopts a structured and analytical viewpoint on source code vulnerability detection. Rather than proposing a single algorithm and claiming universal superiority, the paper focuses on organizing the problem space and providing a coherent comparison that supports both research understanding and practical decision making. The main contributions of this paper are threefold:

(i) **Vulnerability classification by causes and technical characteristics:** The paper develops a classification approach that groups vulnerabilities according to their causes and technical manifestations, thereby clarifying the nature and detection challenges of each group, including memory-related vulnerabilities[8] and input-handling vulnerabilities [9].

(ii) **Evaluation and comparison of major detection method categories:** The paper analyzes representative method groups, static analysis, dynamic analysis/fuzzing [10], and learning-based approaches [11], and compares them using consistent qualitative criteria, highlighting their strengths, limitations, and applicability

boundaries across vulnerability types.

(iii) **Hybrid-oriented direction and proposed approach discussion:** Based on the comparative findings, the paper discusses a combined approach that leverages complementary advantages of different techniques, aligning with recent hybrid-oriented research trends [12]. The experimental part is used to illustrate feasibility and to support the evaluation and comparison statements, rather than aiming at large-scale optimization or state-of-the-art claims.

Through this structured approach, the paper aims to provide researchers and software engineers with a systematic reference for evaluating, selecting, and orienting the development of vulnerability detection solutions in practice.

The remainder of this paper is organized as follows: Section 2 presents a structured classification of source code vulnerabilities based on causes, technical characteristics, development stages, and application contexts; Section 3 analyzes major categories of vulnerability detection methods, including static analysis, dynamic analysis, and learning-based approaches; Section 4 provides a qualitative comparative evaluation framework that highlights strengths and limitations of each method group; Section 5 introduces an illustrative hybrid detection model and presents baseline experimental validation to support the analytical findings; Section 6 discusses implications, limitations, and practical considerations; Finally, Section 7 concludes the paper and outlines directions for future work.

2. SOURCE CODE VULNERABILITY CLASSIFICATION

Source code vulnerabilities appear in diverse forms due to differences in causes, technical characteristics, and exploitation contexts. Since each type exhibits distinct structural and behavioral properties, treating vulnerabilities as a homogeneous group limits effective analysis and evaluation of detection methods.Vulnerability classification clarifies the relationship between vulnerability characteristics and detection capabilities, providing a structured basis for comparing different approaches. It therefore serves as a foundational step for the analytical and comparative discussions presented in this paper.

2.1. Classification Based on Causes and Technical Characteristics

Classifying source code vulnerabilities by their causes and technical characteristics is a common approach in software security research. This perspective clarifies the relationship between code structure, runtime behavior, and exploitability, allowing vulnerabilities to be grouped into several main categories as follows:

- **Memory management related vulnerabilities [13]:**

Memory management vulnerabilities occur when a program performs unsafe operations on memory regions, leading to data overwriting or unauthorized memory access. Common forms include buffer overflow, out-of-bounds access, and use-after-free. This group of vulnerabilities is frequently encountered in programming languages that allow manual memory management, such as C and C++. The following example illustrates a buffer overflow case caused by the lack of input length control:

```
void copy(char *input) {
   char buf[16];
strcpy(buf, input);}
```

In the above code snippet, if the input data exceeds the size of buf, adjacent memory regions may be overwritten. Detecting this type of vulnerability becomes difficult when the program contains multiple conditional branches or depends on dynamic input data, making it challenging for conventional testing techniques to achieve sufficient coverage.

- **Input handling related vulnerabilities [14]:**

This group of vulnerabilities arises when a program processes data from untrusted sources without performing necessary validation or constraints. Typical vulnerabilities include injection, format string vulnerabilities, and errors during data deserialization. A common characteristic of this group is that input data can alter the intended behavior of the program. The following example illustrates an injection vulnerability when constructing a query from user-provided data in Java:

```
String query = "SELECT * FROM users WHERE id = " + userInput;
Statement stmt = conn.createStatement();
stmt.executeQuery(query);
```

In this case, if userInput contains intentionally injected data, the executed query may be modified. Although the nature of this vulnerability is relatively clear, automated detection is not always straightforward because data validity depends on the usage context and application logic.

- **Logic and control flow related vulnerabilities [15]:**

Logic vulnerabilities arise when a program performs incorrect or missing necessary checks within its processing flow. Unlike purely technical vulnerabilities, this group typically does not cause runtime errors but instead leads to unintended behavior under specific scenarios. The following example illustrates a logic flaw in access control within Python code:

```
def withdraw(amount, balance):
    if amount < balance:
        balance -= amount
    return balance
```

In the above code snippet, the absence of a check for the condition amount > 0 may result in behaviors that deviate from the intended business objectives. Because such logic vulnerabilities do not trigger execution errors, they are very difficult to detect using syntactic analysis or error-based testing, and they often require an understanding of the system context and functional requirements.

- **Vulnerabilities caused by unsafe API or library usage [16]:**

This group of vulnerabilities is related to the use of functions, APIs, or external libraries in an unsafe manner or in ways that are inappropriate for the given context. Some functions or APIs, although syntactically valid, may pose risks if they are used improperly. The following example illustrates the use of an unsafe string handling function in C code:

```
char buf[32];
gets(buf);
```

Although the above code snippet is simple, the use of the gets function can lead to buffer overflow if the input data exceeds the allocated memory size. The difficulty in detecting this type of vulnerability lies in the fact that not every API call is dangerous; rather, the level of risk strongly depends on how and in what context the API is used within the program.

From the vulnerability groups discussed above, it can be observed that each type of vulnerability exhibits distinct technical characteristics and detection challenges. This diversity indicates that no single method can effectively detect all types of source code vulnerabilities, thereby emphasizing

the need to combine multiple approaches in modern vulnerability detection research and systems.

2.2. Classification Based on Stages in The Software Development Life Cycle

In addition to cause-based classification, source code vulnerabilities can be analyzed according to the stage of the software development life cycle at which they arise. This perspective helps link vulnerabilities to appropriate detection methods and supports more effective method selection and evaluation.

- **Design-level vulnerabilities [17]:**

Design-level vulnerabilities emerge at the early stages of the software development life cycle, when system architecture, authorization models, or business process flows are defined. Vulnerabilities in this group are typically related to inadequately designed authentication and authorization mechanisms, improper functional separation, or the lack of security constraints at the architectural level. Since they exist before concrete source code is implemented, design-level vulnerabilities are often difficult to identify through pure source code inspection.

Detecting vulnerabilities at the design level requires analyzing system architecture and initial security assumptions, rather than relying solely on program syntax or structure. If not identified early, these vulnerabilities may propagate into the implementation stage and become more difficult to remediate in later phases of the software life cycle.

- **Implementation-level vulnerabilities [18]:**

Implementation-level vulnerabilities arise during the programming phase, when designs are translated into concrete source code. This group includes common issues such as unsafe memory management, insufficient input validation, errors in condition checking, and improper use of APIs and libraries. It is the most extensively studied group of vulnerabilities and is often directly related to the structure and content of source code.

Because they occur at the implementation level, these vulnerabilities can be detected through source code inspection techniques or by observing program execution behavior. However, the effectiveness of detection strongly depends on the characteristics of each vulnerability type, as well as the coverage capability of the applied method.

- **Integration and deployment vulnerabilities [19]:**

Integration and deployment vulnerabilities arise when software components are combined or deployed in real-world environments. Although individual modules may appear secure in isolation, incompatibilities, misconfigurations, or violated assumptions during integration can introduce new weaknesses.

Detecting such vulnerabilities typically requires analyzing the system in its operational context, where runtime interactions and deployment conditions are considered. This life-cycle-based classification highlights that detection methods should be selected according to the stage at which vulnerabilities emerge, forming a basis for the comparative analysis in Section 3.

2.3. Classification Based on Context and Application Environment

Beyond technical causes and development stages, the application context also influences how

vulnerabilities arise, their risk level, and their detectability. The same vulnerability may have different security implications across environments. Therefore, environment-based classification provides a more comprehensive view of vulnerability detection and evaluation challenges.

- **Vulnerabilities in web applications [20]:**

In web applications, source code vulnerabilities commonly appear at points where user data is received and processed. Errors related to input handling, authentication, or session management can lead to serious risks, as such systems are typically publicly accessible. The following example illustrates a case of unsafe input handling in a web application:

```
String query = "SELECT * FROM users WHERE name = '" + userInput + "'";
statement.executeQuery(query);
```

In a web environment, the above code snippet can be exploited to alter the executed query. Detecting this vulnerability typically requires examining input data flows, the relationships among server-side components, and user interaction scenarios, rather than relying solely on source code syntax inspection.

- **Vulnerabilities in embedded and IoT systems [21]:**

In embedded and IoT systems, source code vulnerabilities are often associated with resource constraints and simplified access control mechanisms. Errors in memory management or unsafe data handling can lead to serious consequences, as such devices typically operate continuously and are difficult to update. The following example illustrates a memory management flaw in an embedded program:

```
void receive(char *data) {
   char buf[32];
strcpy(buf, data);}
```

Although this flaw is similar in nature to vulnerabilities found in conventional applications, in an IoT environment it may lead to device takeover or disruption of the entire system's operation. Vulnerability detection in this case often relies heavily on source code analysis and behavioral modeling, since dynamic testing on real devices is difficult to perform.

- **Vulnerabilities in cloud services and microservices architectures [22]:**

In cloud and microservices environments, vulnerabilities may arise not only within individual services but also from their interactions. Inconsistent authentication or authorization assumptions across components can create weaknesses that are difficult to detect. For example:

```
@app.route("/internal/data")
def get_data():
return sensitive_data
```

While this code may appear safe in isolation, it can become vulnerable when exposed beyond its intended scope within a distributed system. Detecting such issues requires considering deployment architecture, communication flows, and configuration settings rather than evaluating services independently.

These examples show that vulnerability risk and detection requirements depend heavily on deployment context, emphasizing the need to select and combine appropriate detection methods in modern systems.

2.4. Relationship Between Vulnerability Types and Detection Methods

The nature of source code vulnerabilities directly influences the effectiveness of detection methods. Therefore, evaluation should consider specific vulnerability groups rather than

assessing techniques in isolation.

- Memory-related vulnerabilities (e.g., buffer overflow, out-of-bounds access) are closely linked to code structure and memory operations, making static analysis with data and control flow examination particularly suitable, although false positives remain a concern.
- Logic and control-flow vulnerabilities are harder to detect using rule-based approaches because they depend on design intent and business context. Machine learning and deep learning methods are more promising in this area due to their ability to learn complex behavioral patterns.
- Vulnerabilities involving unsafe API or library usage require context-aware analysis, as the same function may be safe or unsafe depending on usage conditions. Semantic representation models and contextual reasoning are therefore essential for accurate assessment.

Overall, no single method can effectively detect all vulnerability types. This relationship highlights the need for hybrid approaches that combine complementary strengths, forming the basis for the comparative analysis in Section 3.

- **The above analysis shows that:**

(i) Memory, input handling, and logic vulnerabilities remain the most common in current systems. Memory-related issues are prevalent in low-level languages such as C/C++, while input and access control flaws are frequent in web and server-side applications, showing that vulnerability distribution depends on programming languages and deployment environments.

(ii) Static and dynamic analysis dominate practice due to automation and integration capability. Meanwhile, machine learning and deep learning approaches gain attention for context-dependent vulnerabilities, although their effectiveness varies by vulnerability type and data quality.

(iii) Existing studies often focus on common and data-accessible vulnerabilities, while complex logic flaws and component interaction issues remain challenging.

These observations emphasize the need to evaluate detection methods in relation to specific vulnerability types, forming the basis for the analysis in Section 3.

3. VULNERABILITY DETECTION METHODS

3.1. Static Analysis (SAST)

Static Application Security Testing (SAST) [24] is a widely used approach that analyzes source or intermediate code without execution, relying on predefined rules and vulnerability patterns to detect weaknesses early in development.

- SAST examines code structure using syntactic, control flow, and data flow analysis. Tools typically construct intermediate representations such as abstract syntax trees or control flow graphs and apply rule-based detection models derived from known vulnerability patterns.

For example, the following code may lead to a buffer overflow:

```
void copy(char *input) {
   char buf[32];
strcpy(buf, input);}
```

The unsafe use of strcpy without input length control can be detected by identifying risky function calls combined with buffer size information.

- Advantages. SAST enables early detection, fast analysis, and easy integration into development workflows (e.g., DevSecOps pipelines). It can analyze the entire codebase, including branches difficult to trigger during execution.
- Disadvantages. SAST often produces high false positives and struggles to distinguish safe from unsafe usage in complex systems. It is less effective for vulnerabilities dependent on

execution context or business logic.

Overall, SAST is well suited for structure-related vulnerabilities such as memory errors and unsafe API usage, but limited for context-dependent flaws, highlighting the need for complementary methods.

3.2. Dynamic Analysis (DAST, Fuzzing)

Dynamic Application Security Testing (DAST) [25] is a group of vulnerability detection methods based on observing program behavior during execution. Unlike static analysis, DAST does not directly inspect source code structure but instead focuses on evaluating program responses to different inputs and usage scenarios. This approach is particularly suitable for vulnerabilities that only manifest when the program is executed under specific conditions.

- The principle of DAST is based on supplying designed or automatically generated input data to a running program and then monitoring abnormal signs such as execution errors, program crashes, memory violations, or behaviors that deviate from expectations. Among these techniques, fuzzing is a representative approach that uses random or semi-structured input generators to trigger different execution paths of the program.

Modern fuzzing tools such as AFL (American Fuzzy Lop) [26] and libFuzzer [27] improve detection effectiveness by leveraging feedback from program execution, such as code coverage information, to guide input generation. As a result, fuzzing is not purely random but is capable of focusing on program branches that have not yet been tested.

- The following example illustrates an input processing function that may cause errors when receiving unexpected value:

```
int parse(char *input) {
   int x = atoi(input);
   int arr[10];
   return arr[x];}
```

When the program is executed with different input data, a fuzzing tool may generate a value of x that exceeds the array bounds, leading to a memory access error. Such errors are often difficult to detect using static analysis alone but can be effectively identified through dynamic analysis and fuzzing.

- Advantages: A prominent advantage of DAST is its ability to detect execution dependent vulnerabilities, particularly memory management errors, input handling flaws, and issues that only appear under specific runtime scenarios. Dynamic analysis helps reduce the number of false positives compared to some static analysis techniques, since vulnerabilities are confirmed through actual runtime failures. In addition, modern fuzzing tools offer a high degree of automation and do not require deep knowledge of the internal structure of the program.
- Disadvantages: Despite its effectiveness in many cases, DAST also has significant limitations. This approach strongly depends on the coverage of test scenarios and input data, making it difficult to ensure comprehensive vulnerability detection in large and complex programs. Moreover, DAST often requires a suitable execution environment and incurs higher time costs than static analysis. For logic vulnerabilities or those that depend on business context, triggering faulty behavior through fuzzing or dynamic testing remains a major challenge.

Considering the relationship between vulnerability types and detection methods discussed in Section 2.6, DAST is particularly suitable for vulnerabilities that only manifest during program execution, such as memory errors or input handling flaws. However, the detection capability of this approach is still constrained by execution coverage, indicating that DAST needs to be combined with other approaches to achieve more comprehensive effectiveness.

3.3. Machine Learning-based Vulnerability Detection

Traditional machine learning approaches for source code vulnerability detection [28] aim to overcome the limitations of rule-based analysis by learning from labeled source code data. Instead of relying solely on predefined patterns, these models distinguish between secure and vulnerable code through data-driven training, enabling better adaptation to diverse vulnerabilities encountered in practice.

- Machine learning-based approaches typically involve two stages: feature extraction and classifier training. Source code is first transformed into numerical representations capturing syntactic, structural, or behavioral properties, then classifiers such as SVM [29], Random Forest [30], or XGBoost [31] are trained to distinguish vulnerable from non-vulnerable samples.
- Model effectiveness depends largely on feature quality. Common feature types include syntactic features from ASTs [32], structural features from control flow graphs (CFGs) [33] and program dependence graphs (PDGs) [34], and low-level behavioral or statistical features. The choice and combination of these features directly affect detection capability.
- For example, buffer overflow detection in C programs may rely on AST and CFG features describing memory operations, which are then classified using models such as Random Forest based on learned patterns.
- Advantages: ML approaches reduce reliance on handcrafted rules and can capture nonlinear relationships among features. Models like RF and XGBoost often improve detection performance and allow feature importance analysis for partial interpretability.
- Disadvantages: Performance depends heavily on labeled data quality and feature engineering. Extracting AST, CFG, or PDG features incurs high preprocessing cost and scalability challenges. Moreover, traditional ML models still struggle with logic vulnerabilities and context-dependent semantics.

Considering the relationship between vulnerability types and detection methods discussed in Section 2.6, traditional machine learning approaches are suitable for vulnerabilities that can be modeled through syntactic and structural features. However, for vulnerabilities that require deep understanding of program semantics and context, these methods need to be combined with or extended by deep learning techniques, which are presented in the next section.

3.4. Deep Learning-based Transformer-based Vulnerability Detection

Deep learning approaches for source code vulnerability detection [35] aim to overcome the limitations of traditional machine learning by learning representations directly from source code rather than relying on handcrafted features. Transformer-based models [36] have recently become a prominent direction due to their ability to capture semantic and contextual relationships.

- Deep learning approaches map source code into latent representations using multi-layer neural networks for vulnerability classification. Early models employed RNNs or CNNs, while Transformer-based architectures leverage self-attention [37] to capture long-range dependencies and overcome contextual limitations of previous methods.
- In practice, Transformer models are pre-trained on large code corpora and fine-tuned for vulnerability detection tasks, enabling them to learn general syntactic and semantic knowledge before adapting to specific vulnerabilities.
- Unlike traditional ML, deep learning models do not require manual feature extraction. Source code is represented as token sequences or combined with structural information such as ASTs and data flows. Transformer representations capture semantic and structural relationships, supporting deeper contextual understanding.

- For example, in detecting access-control logic flaws, a Transformer model can learn dependencies among conditional checks, state variables, and function calls across an entire function without predefined rules.
- Advantages: Transformer-based models effectively capture complex semantics and context, making them suitable for logic or interaction-based vulnerabilities. Pre-training reduces feature engineering effort and improves generalization.
- Disadvantages: High computational cost, large resource requirements, limited interpretability, and dependence on training data remain key challenges, particularly for deployment in large-scale or cross-project scenarios.

Considering the relationship between vulnerability types and detection methods discussed in Section 2.6, deep learning and Transformer-based approaches are particularly suitable for logic vulnerabilities and cases that require deep understanding of program semantics. However, due to cost and interpretability limitations, these approaches are often most effective when combined with other methods within a comprehensive vulnerability detection system.

3.5. LLM-based/Hybrid Approaches

LLM-based approaches [38] and hybrid models [39] aim to leverage large-scale semantic reasoning capabilities while addressing the limitations of individual detection methods. Rather than treating vulnerability detection purely as a classification task, these approaches integrate multiple techniques within a unified pipeline.

Representative source-code-oriented models include CodeLLaMA [40], StarCoder [41], and GPT-4 [42]. CodeLLaMA is suitable for research due to its specialization and independent deployment capability; StarCoder provides an open-source multilingual alternative; GPT-4 serves as a reasoning benchmark despite cost and reproducibility limitations.

LLM-based methods utilize pre-trained models to analyze and reason about code, either directly or through task-specific fine-tuning. In hybrid systems, LLMs are typically integrated with static analysis, dynamic analysis, or machine learning. Alerts generated by SAST or ML models can be further examined by LLMs to reassess severity and reduce false positives.

Unlike earlier deep learning models, LLMs capture syntax, semantics, and global context simultaneously. For example, suspicious access-control code identified by static analysis can be re-evaluated by an LLM to determine whether it represents a genuine vulnerability or a legitimate usage pattern.

The main advantage of LLM-based and hybrid approaches lies in their semantic reasoning capability, particularly for logic and context-dependent vulnerabilities. However, high computational cost, deployment complexity, and stability concerns limit their standalone use. Therefore, LLMs are most effective as complementary components within a broader detection architecture rather than as full replacements for traditional methods.

3.6. Proposed Source Code Vulnerability Detection Architecture

The analyses in Sections 3.1-3.5 indicate that each detection method is effective only within certain scopes. Static analysis provides early detection but often generates high false positives; dynamic analysis and fuzzing validate runtime behavior but depend on test coverage; machine learning and deep learning improve detection of complex patterns but rely on data quality and computational resources; large language models offer strong semantic reasoning but face challenges in stability and deployment efficiency. These limitations suggest that no single method

can fully satisfy practical detection requirements.

Based on this insight, a hybrid-oriented architecture is proposed, organizing complementary methods into functional layers rather than replacing existing techniques.

As illustrated in Figure 1, the pipeline includes:

(i) Static analysis for early-stage screening and broad coverage;
(ii) Dynamic analysis and fuzzing to validate execution-level vulnerabilities and reduce false positives;
(iii) Machine learning/deep learning models to analyze syntactic and structural patterns, particularly for complex or logic-related vulnerabilities;
(iv) Large language model-based reasoning for semantic reassessment and alert consolidation, helping reduce noise in final results.

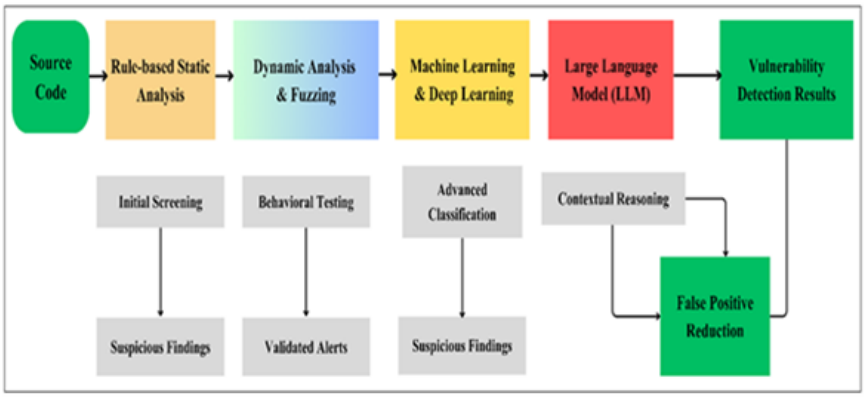

Figure 1. Pipeline for source code vulnerability detection combining static analysis, dynamic analysis, machine learning, and large language models

The core idea lies in coordinated layering, allowing flexible adjustment according to accuracy requirements, computational constraints, and development stages. Although not fully implemented within this study, the architecture outlines a practical direction for future hybrid vulnerability detection systems, consistent with the paper's analytical and comparative objectives.

4. Evaluation and Comparison of Source Code Vulnerability Detection Methods

This section evaluates and compares the detection methods discussed in Section 3 to clarify their application scope, strengths, and limitations. The analysis provides a structured perspective on the overall landscape of source code vulnerability detection.

4.1. Evaluation Criteria

The comparison of methods is conducted based on qualitative criteria that reflect common requirements in both research and practical deployment, including: (i) Suitable vulnerability types: the groups of vulnerabilities that a method can effectively detect; (ii) Detection capability: the degree to which target vulnerabilities are correctly identified; (iii) False positives: the tendency to generate alerts that do not correspond to actual vulnerabilities; (iv) Computational and deployment cost: the resources required to operate the method; (v) Scalability: the suitability of the method when applied to large-scale or multilingual systems; and (vi)Context dependence:

the ability to handle vulnerabilities that depend on program logic and execution context.
These criteria are applied consistently throughout the comparison tables and discussion to ensure coherence in the evaluation.

4.2. Summary Comparison Table of Detection Methods

Table 1. Qualitative comparison of source code vulnerability detection methods

Method	Suitable vulnerability types	Detection capability	False positives	Cost	Context depen-dence
SAST [24]	Memory, API, syntax	Medium	High	Low	Low
DAST/Fuzzing [25]	Memory, input handling	Medium - High	Low	Medium - High	Medium
ML-based [28]	Pattern-based vulnerabilities	Medium	Medium	Medium	Medium
DL/Transformer [36]	Logic, semantics	High	Medium	High	High
LLM-based [38]	Logic, complex context	High	Low - Medium	Very high	Very high
Hybrid [39]	Multiple vulnerability types	High	Low	High	High

The comparison indicates that no single method is superior across all criteria; each approach has distinct strengths and limitations.

Transformer-based models have become central to modern deep learning due to their ability to capture long-range dependencies through self-attention. Variants such as BERT [43], CodeBERT [44], and GraphCodeBERT [45] extend this architecture to better represent source code by incorporating syntactic and structural information. More recently, large-scale models such as CodeLLaMA further enhance semantic reasoning, providing a foundation for hybrid approaches that integrate deep learning and large language models in vulnerability detection.

4.3. Analysis and Discussion

The comparison highlights distinct strengths and limitations across detection methods.

- SAST remains widely used due to low cost and easy integration into development workflows. However, its reliance on fixed rules leads to high false positives and limited effectiveness for context-dependent or logic vulnerabilities.
- DAST and fuzzing are effective for vulnerabilities that manifest during execution, particularly memory and input-handling flaws. With sufficient coverage, they can improve alert accuracy, but execution cost and limited coverage remain major constraints.
- Machine learning and deep learning methods extend detection to pattern-based and semantic vulnerabilities. Transformer-based models, in particular, show strong capability in handling logic-related cases. Nevertheless, these approaches require high-quality data, significant computational resources, and often lack interpretability.
- LLM-based approaches further enhance semantic reasoning and global context understanding, helping reduce false positives and detect complex vulnerabilities. However, high deployment cost and concerns about stability limit their standalone use.
- Hybrid approaches attempt to balance coverage and precision by combining complementary strengths. Their effectiveness depends on careful system design and integration,

typically at higher operational cost.
Overall, the analysis confirms that no single method is universally effective. Detection capability depends on vulnerability type, deployment context, and system requirements. A unified comparative framework clarifies this landscape and explains the growing interest in hybrid strategies.

5. PROPOSED MODEL AND EXPERIMENTAL EVALUATION

5.1. Hybrid Vulnerability Detection Model Combining ML and LLM

Based on the comparative analysis in Sections 3 and 4, this paper proposes a hybrid source code vulnerability detection model that combines traditional machine learning with a large language model to leverage complementary strengths at different stages of detection. CodeLLaMA-34B is employed at the triage stage to provide semantic and context-aware reasoning for suspicious code fragments.

The architecture follows a two-layer design:

- **First layer: Machine Learning Classification:** Traditional models such as Random Forest, XGBoost, and SVM are used to perform fast classification based on syntactic and structural features. This layer serves as an initial filtering stage, enabling efficient large-scale processing. In experiments, ML models achieved F1-scores of 0.72-0.78, indicating stable detection capability for common vulnerability types.

- **Second layer: LLM-based Triage:** Alerts generated by the ML layer are passed to CodeLLaMA-34B for semantic analysis. Instead of analyzing the entire codebase, the LLM processes only suspicious cases, reducing computational cost. The triage stage examines logical context and code intent to reassess alerts and reduce false positives. Experimental results show that LLM-based triage reduces the average False Positive Rate by approximately 25-35% compared with standalone ML models.

The model is implemented at a baseline level, including preprocessing, ML classification, and LLM-based triage. The objective is not performance optimization but feasibility validation. Results indicate that the hybrid model maintains detection performance while significantly improving alert precision through noise reduction, supporting the practical value of integrating machine learning and semantic reasoning in a unified architecture.

5.2. Experimental setup

The experiments aim to provide illustrative validation rather than state-of-the-art benchmarking. The hybrid ML-LLM model is evaluated to examine whether semantic triage can reduce false positives while maintaining detection capability. The results should therefore be interpreted as empirical support for the hybrid rationale derived from the comparative analysis.

- **Datasets.** Experiments are conducted on two widely used benchmarks: The Juliet Test Suite [46] and Big-Vul [47]. Juliet contains synthetically generated C/C++ functions with labeled vulnerable and non-vulnerable cases, while Big-Vul consists of real-world vulnerable and patched functions from open-source projects.
- **Dataset clarification.** No new dataset is constructed. Both datasets are publicly available and used in their standard form. The unit of analysis is function-level code. For Big-Vul, vulnerable and corresponding non-vulnerable functions are treated as independent samples

after preprocessing. No manual relabeling is performed; preprocessing includes cleaning, normalization, and feature extraction for machine learning.

- **Baselines.** Three configurations are compared: (i) SAST only (rule-based static analysis); (ii) ML only (Random Forest and XGBoost); (iii) ML + LLM-based triage (proposed hybrid model).
- **Evaluation metrics.** Performance is measured using Precision, Recall, F1-score, and False Positive Rate, emphasizing detection quality and alert noise reduction.

5.3. Experimental results and analysis

Table 2 presents an overall comparison between SAST, traditional machine learning models, and the proposed hybrid ML-LLM models on the Juliet Test Suite and Big-Vul datasets. Integrating CodeLLaMA-34B at the triage stage significantly reduces the False Positive Rate while preserving detection performance.

SAST achieves high Recall (0.81) but suffers from low Precision (0.48) and a high False Positive Rate (0.42), reflecting the limitations of rule-based analysis. Traditional machine learning improves performance, with Random Forest and XGBoost reaching F1-scores of 0.71 and 0.74, and reducing the False Positive Rate to 0.30 and 0.26, respectively. XGBoost consistently outperforms Random Forest due to its stronger nonlinear modeling capability.

Table 2. Experimental results comparing source code vulnerability detection methods

Method	Precision	Recall	F1-score	False Positive Rate
SAST	0.48	0.81	0.60	0.42
ML (Random Forest)	0.68	0.74	0.71	0.30
ML (XGBoost)	0.72	0.76	0.74	0.26
Hybrid (RF + CodeLLaMA-34B)	0.80	0.75	0.77	0.20
Hybrid (XGBoost + CodeLLaMA-34B)	0.82	0.76	0.79	0.18

The hybrid models further enhance detection quality. RF + CodeLLaMA-34B achieves an F1-score of 0.77 and a False Positive Rate of 0.20, while XGBoost + CodeLLaMA-34B reaches the best overall performance (F1 = 0.79, FPR = 0.18). These results confirm that semantic triage effectively reduces alert noise without sacrificing detection capability.

Table 3. Experimental results of the Hybrid model on individual datasets

Data Set	Method	Precision	Recall	F1-score	False/Positive Rate
Juliet Test Suite	RF + CodeLLaMA-34B	0.83	0.77	0.80	0.18
	XGBoost + CodeLLaMA-34B	0.86	0.78	0.82	0.15
Big-Vul	RF + CodeLLaMA-34B	0.77	0.74	0.75	0.22
	XGBoost + CodeLLaMA-34B	0.80	0.75	0.77	0.20

Table 3 provides dataset-level results. On the Juliet Test Suite, the hybrid models achieve F1-scores of 0.80-0.82, with XGBoost + CodeLLaMA-34B performing best (Precision = 0.86, FPR = 0.15). On Big-Vul, performance decreases slightly (F1 = 0.75-0.77; FPR = 0.20-0.22), but remains stable, indicating reasonable generalization to more realistic and diverse source code.

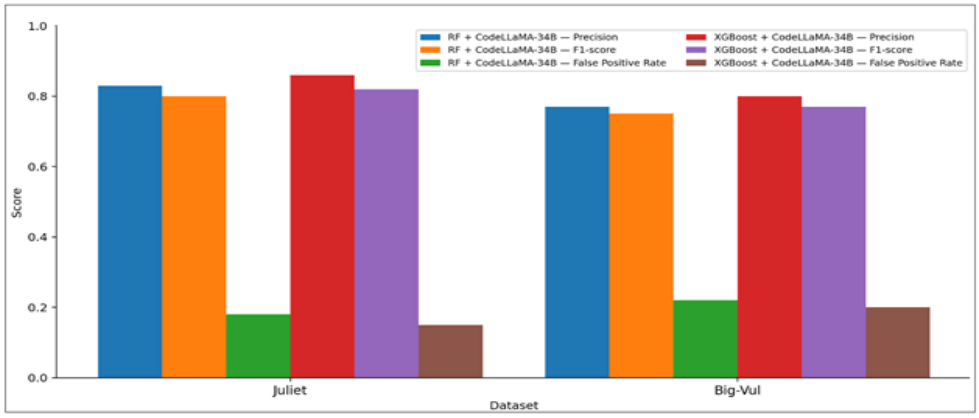

Figure 2. Comparison of Precision, F1-score, and False Positive Rate of the hybrid models (RF + CodeLLaMA-34B and XGBoost + CodeLLaMA-34B) on the Juliet Test Suite and Big-Vul

On the Big-Vul dataset, both hybrid models show slightly lower performance due to the more realistic and noisier nature of the code. However, they maintain stable F1-scores and low False Positive Rates, indicating good generalization to real-world scenarios.

As illustrated in Figure 4, the XGBoost + CodeLLaMA-34B model consistently outperforms RF + CodeLLaMA-34B, achieving higher Precision and F1-score while maintaining a lower False Positive Rate across both Juliet and Big-Vul datasets.

Although performance on Big-Vul is slightly below that of Juliet, both hybrid models effectively control alert noise, demonstrating the benefit of LLM-based triage. Among CodeLLaMA variants (7B, 13B, and 34B), CodeLLaMA-34B is selected due to its stronger semantic reasoning capability, making it more suitable for the triage role in the proposed hybrid architecture.

6. DISCUSSION

The experimental results in Section 5 provide several key observations regarding the effectiveness of different detection methods and the value of the proposed hybrid approach.

- First, rule-based static analysis (SAST) achieves high Recall (above 0.80), reflecting strong coverage and early detection capability. However, its low Precision (≈0.48) and high False Positive Rate (above 0.40) confirm its limitations in handling context-dependent and logic-related vulnerabilities.
- Traditional machine learning models, including Random Forest and XGBoost, significantly improve detection quality. The F1-score increases from approximately 0.60 (SAST) to 0.71-0.74, while the False Positive Rate decreases to 0.26-0.30. Nevertheless, relying solely on syntactic and structural features remains insufficient to fully eliminate false positives.
- Integrating a large language model at the triage stage further enhances performance. The hybrid models reduce the False Positive Rate by approximately 25-35% compared to standalone ML models, while maintaining or slightly improving the F1-score (0.77-0.79). Using CodeLLaMA-34B improves Precision to above 0.80, demonstrating the benefit of semantic reasoning for distinguishing context-dependent cases.
- Between the two hybrid configurations, XGBoost + CodeLLaMA-34B consistently outperforms RF + CodeLLaMA-34B, achieving F1-scores approximately 0.02-0.03 higher and lower False Positive Rates (around 0.15-0.20 versus 0.18-0.22). This indicates the advantage of XGBoost in capturing nonlinear feature relationships while confirming the robustness of the

hybrid architecture.

- Across datasets, the hybrid model achieves F1-scores of 0.80-0.82 on the Juliet Test Suite and 0.75-0.77 on Big-Vul. Although performance slightly decreases on Big-Vul due to its realistic and noisier nature, maintaining a False Positive Rate below 0.22 demonstrates stable generalization capability.

Despite these positive results, the study remains limited to baseline experiments on a small number of datasets. Computational cost and interpretability of LLM-based triage are not analyzed in depth and should be addressed in future work.

Overall, the findings empirically support the hybrid ML-LLM rationale and align with the comparative analysis presented in earlier sections.

7. CONCLUSION

This paper presents a systematic perspective on source code vulnerability detection, beginning with a structured classification of vulnerability types and followed by a comparative analysis of major detection approaches. The comparison clarifies the strengths, limitations, and applicability of each method group.

Based on this analysis, a hybrid detection model combining traditional machine learning with a large language model at the triage stage is introduced and evaluated on the Juliet Test Suite and Big-Vul datasets. The results demonstrate that the hybrid approach reduces the false positive rate while maintaining or improving the F1-score, confirming the feasibility of integrating complementary methods.

Although limited to baseline evaluation, the findings support the complementary role of machine learning and large language models in vulnerability detection. Future work may extend the evaluation to more diverse datasets, analyze computational cost in greater depth, and enhance interpretability through Explainable Artificial Intelligence techniques.

REFERENCES

[1] E. Kanaan, S. S. Alam and M. S. Akter, "Survey of Machine Learning Techniques for Detecting Buffer Overflow Vulnerabilities," Apr. 30 2025, doi:10.13140/RG.2.2.22978.08640.

[2] R. Li, B. Zhang and C. Tang, "Identifying Exploitable Memory Objects for Out-of-Bound Write Vulnerabilities," *Electronics Letters*, vol. 60, no. 5, pp. 1-4, Feb. 2024, doi:10.1049/ell2.13136.

[3] X. Zhao, H. Qu, J. Yi, J. Wang, M. Tian, and F. Zhao, "A fuzzer for detecting use-after-free vulnerabilities," Mathematics, vol. 12, no. 21, Art. no. 3431, Nov. 2024, doi:10.3390/math12213431.

[4] L. Aburashed, M. A. Almoush, and W. Alrefai, "SQL injection attack detection using machine learning algorithms," Semarak International Journal of Machine Learning, vol. 2, no. 1, pp. 112–120, Jun. 2024, doi:10.37934/sijml.2.1.112.

[5] X. Yin, R. Cai, Y. Zhang, L. Li, Q. Yang, and S. Liu, “Accelerating Command Injection Vulnerability Discovery in Embedded Firmware with Static Backtracking Analysis,” in Proceedings of the 12th International Conference on the Internet of Things (IoT ’22), 2023, pp. 65–72, doi: 10.1145/3567445.3567458.

[6] S. Xu, Z. Jiang, and P. Xie, “FormatAEG: A framework for bypassing ASLR defense and automated exploitation of format string vulnerability,” The Computer Journal, vol. 67, no. 12, Oct. 2024, Art. no. bxae069, doi: 10.1093/comjnl/bxae069.

[7] H. Yang, X. Gu, Z. Cui, et al., "CrossFuzz: Cross-contract fuzzing for smart contract vulnerability detection," Science of Computer Programming, vol. 233, Jan. 2024, Art. no. 103076, doi: 10.1016/j.scico.2023.103076.

[8] S. Cao, X. Sun, W. Liu et al., “Learning to detect memory-related vulnerabilities,” ACM Transactions on Software Engineering and Methodology, vol. 33, no. 1, pp. 1–37, Jan. 2024.

[9] C. Brandi, G. Perrone, and S. Romano, “Sniping at web applications to discover input-handling

vulnerabilities," Journal of Computer Virology and Hacking Techniques, vol. 20, no. 2, Apr. 2024, pp. 1–18, doi: 10.1007/s11416-024-00518-0.

[10] R. Singh, M. Gupta, and S. M. Patil, "Analysis of web application vulnerabilities using dynamic application security testing," in Proc. 2024 IEEE 9th International Conference for Convergence in Technology (I2CT), Pune, India, Apr. 2024, doi:10.1109/I2CT61223.2024.10543484.

[11] M. Bresil, P. Prasad, and U. Bukar, "Deep learning-based vulnerability detection solutions in smart contracts: A comparative and meta-analysis of existing approaches," IEEE Access, vol. 13, pp. 1–24, 2025, doi: 10.1109/ACCESS.2025.3532326.

[12] Q. Yu, "From rule-driven to data-driven: Technological evolution, challenges, and future trends in smart contract vulnerability detection," Transactions on Computer Science and Intelligent Systems Research, Jul. 2025, doi: 10.62051/c9jgtf18.

[13] S. Cao, X. Sun, C. Tao, et al., "MVD: Memory-related vulnerability detection based on flow-sensitive graph neural networks," in Proc. Int. Conf. on Software Engineering and Knowledge Engineering (SEKE), Redwood City, CA, USA, Mar. 2022, doi:10.1145/3510003.3510219.

[14] S. Cao, B. He, X. Sun, Y. Ouyang, C. Zhang, X. Wu, T. Su, L. Bo, B. Li, C. Ma, J. Li, and T. Wei, "ODDFUZZ: Discovering Java deserialization vulnerabilities via structure-aware directed greybox fuzzing," arXiv preprint arXiv:2304.04233, 2023.

[15] V. Felmetsger, L. Cavedon, C. Kruegel, and G. Vigna, "Toward Automated Detection of Logic Vulnerabilities in Web Applications,"19th USENIX Security Symposium (USENIX Security 10), Washington, DC, USA, 2010.

[16] Z. Mousavi, C. Islam, M. A. Babar, A. Abuadbba, and K. Moore, "Detecting Misuses of Security APIs: A Systematic Review," ACM Computing Surveys, vol. 57, no. 12, pp. 1-39, 2025.

[17] Q. Rouland, B. Hamid, and J. Jaskolka, "A model-driven formal methods approach to software architectural security vulnerabilities specification and verification," J. Syst. Softw., vol. 219, 112219, 2025, doi: 10.1016/j.jss.2024.112219.

[18] S. Rehman and K. Mustafa, "Research on software design level security vulnerabilities, " ACM SIGSOFT Software Engineering Notes, vol. 34, no. 6, pp. 1-5, Dec. 2009.

[19] I. E. Kezron, "Securing the AI supply chain: Mitigating vulnerabilities in AI model development and deployment," World Journal of Advanced Research and Reviews, vol. 22, no. 2, pp. 1394-1403, May 2024, doi:10.30574/wjarr.2024.22.2.1394.

[20] K. Lian, L. Zhang, M. Yang, et al., "Careless Retention and Management: Understanding and Detecting Data Retention Denial-of-Service Vulnerabilities in Java Web Containers," in Proc. USENIX Security Symposium, 2025.

[21] N. Neshenko, E. Bou-Harb, J. Crichigno, G. Kaddoum, and N. Ghani, "Demystifying IoT Security: An Exhaustive Survey on IoT Vulnerabilities and a First Empirical Look on Internet-Scale IoT Exploitations," IEEE Communications Surveys & Tutorials, vol. 21, no. 3, pp. 2702-2733, 2019, doi:10.1109/COMST.2019.2910750.

[22] S. Duggirala and D. R. P. Singh, "Securing Microservices in the Cloud: Addressing Emerging Threats and Vulnerabilities," International Journal of Research in Modern Engineering & Emerging Technology, vol. 13, no. 6, 2025, doi:10.63345/ijrmeet.org.v13.i6.16.

[23] K. Karthik, S. Singh, M. Mohana, et al., "Temporal Analysis and Common Weakness Enumeration (CWE) Code Prediction for Software Vulnerabilities Using Machine Learning," in Proc. 8th Int. Conf. on Computational System and Information Technology for Sustainable Solutions (CSITSS), 2024, doi:10.1109/CSITSS64042.2024.10816791.

[24] V. Farrapo, E. Rodrigues, and J. C. Machado, "Evaluating the Use of Open-Source and Standalone SAST Tools for Detecting Vulnerabilities in C/C++ Projects," in Proc. Int. Conf. on Software Technologies, 2025, doi:10.5220/001348350003929.

[25] S. Millar, D. Podgurski, P. Miller, et al., "Optimising Vulnerability Triage in DAST with Deep Learning," in Proc. 15th ACM Workshop on Artificial Intelligence and Security (AISec), co-located with CCS, 2022, pp. 1-12, doi:10.1145/3560830.3563724.

[26] S. Godboley, K. Gupta, and M. Rani Golla, "AV-AFL: A Vulnerability Detection Fuzzing Approach by Proving Non-reachable Vulnerabilities using Sound Static Analyser," in Proc. Int. Conf. on Software Technologies, 2022, doi:10.5220/001103290003176.

[27] R. Liao, X. Yan, K. Zhu, et al., "Balancing validity and vulnerability: Knowledge driven seed generation via LLMs for deep learning library fuzzing," Applied Sciences, vol. 15, no. 19, Art. no. 10396, Sep. 2025, doi: 10.3390/app151910396.

[28] S. Bin Hlayil and S. Lida Xu, "Machine learning based vulnerability detection and classification in

Internet of Things device security," Electronics, vol. 12, no. 18, Art. no. 3927, Sep. 2023.
[29] C. Xiong, "Detection of buffer overflow vulnerability in embedded development programs based on SVM algorithm," in Proc. SPIE, vol. 13069, Art. no. 130690I, Jul. 2025.
[30] C. Singh, V. Vijayalakshmi, and H. Raj, "A machine learning approach for web application vulnerability detection using random forest," International Journal for Research in Applied Science & Engineering Technology (IJRASET), vol. 10, no. 12, pp. 4837-4843, Dec. 2022, doi: 10.22214/ijraset.2022.48397.
[31] M. Zhang, S. Fu, P. Shi, et al., "An XGBoost based vulnerability analysis of smart grid cascading failures under topology attacks," in Proc. IEEE Int. Conf. Smart Grid Communications (SmartGridComm), Oct. 2021, pp. 1-6, doi: 10.1109/SMC52423.2021.9658797.
[32] J. Zhu, H. Ge, and R. Luo, "A method to detect vulnerability of Java source code based on AST and graph attention networks," in Proc. IEEE Int. Conf. on Software Engineering and Service Science (ICSTE), Aug. 2024, pp. 1-6, doi: 10.1109/ICSTE63875.2024.00010.
[33] Y. Wang, J. Zhao, L. Zhu, et al., "Smart contract symbol execution vulnerability detection method based on CFG path pruning," in Proc. Int. Symposium on Reliable Distributed Systems Workshops (SRDSW), Jul. 2023, pp. 1-6, doi: 10.1145/3594556.3594618.
[34] Y. Wang, J. Zhao, L. Zhu, et al., "Smart contract symbol execution vulnerability detection method based on CFG path pruning," in Proc. International Symposium on Reliable Distributed Systems Workshops (SRDSW), Jul. 2023, pp. 1-6, doi: 10.1145/3594556.3594618.
[35] Y. He, G. Lin, F. Li, et al., "Enhancing deep learning vulnerability detection through imbalance loss functions: An empirical study," in Proc. Asia Pacific Symposium on Software Engineering (APSEC), Jul. 2024, pp. 1-8, doi: 10.1145/3671016.3671379.
[36] Y. Cao, Y. Dong, and J. Peng, "Vulnerability detection based on transformer and high quality number embedding," Concurrency and Computation: Practice and Experience, vol. 36, no. 18, Art. no. e8292, Sep. 2024, doi: 10.1002/cpe.8292.
[37] T. Wu, L. Chen, G. Shi, et al., "Self attention based automated vulnerability detection with effective data representation," in Proc. IEEE Int. Conf. on Parallel & Distributed Processing with Applications, Big Data & Cloud Computing, Sustainable Computing & Communications, Social Computing & Networking (ISPA/BDCloud/SocialCom/SustainCom), Sep. 2021, pp. 1-8, doi: 10.1109/ISPA-BDCloud-SocialCom-SustainCom52081.2021.00126.
[38] A. A. Mahyari, "Harnessing the power of LLMs in source code vulnerability detection," in Proc. IEEE Military Communications Conference (MILCOM), Aug. 2024, pp. 1-6.
[39] D. Wang, J. Chen, X. Hu, et al., "SCVD-SA: A smart contract vulnerability detection method based on hybrid deep learning model and self-attention mechanism," in Proc. IEEE Int. Conf. on Software Analysis, Evolution and Reengineering (SANER), Mar. 2024, pp. 1-10.
[40] S. Sultana, S. Afreen, and N. U. Eisty, "Code vulnerability detection: A comparative analysis of emerging large language models," arXiv preprint arXiv:2409.10490, Sep. 2024.
[41] R. Li, L. Ben Allal, Y. Zi, et al., "StarCoder: May the source be with you!" arXiv preprint arXiv:2305.06161, Dec. 2023, doi: 10.48550/arXiv.2305.06161.
[42] J. Bae, S. Kwon, and S. Myeong, "Enhancing software code vulnerability detection using GPT-4o and Claude-3.5 Sonnet: A study on prompt engineering techniques," Electronics, vol. 13, no. 13, Art. no. 2657, Jul. 2024, doi: 10.3390/electronics13132657.
[43] J. Haurogné, N. Basheer, and S. Islam, "Vulnerability detection using BERT based LLM model with transparency obligation practice towards trustworthy AI," Machine Learning with Applications, vol. 16, Art. no. 100598, Nov. 2024, doi: 10.1016/j.mlwa.2024.100598.
[44] Y. Xia, H. Shao, and X. Deng, "VulCoBERT: A CodeBERT based system for source code vulnerability detection," in Proc. Int. Conf. on Generative Artificial Intelligence and Information Security (GAIIS), May 2024, pp. 1-6, doi: 10.1145/3665348.3665391.
[45] R. Wang, S. Xu, S. Jiang, et al., "SCL-CVD: Supervised contrastive learning for code vulnerability detection via GraphCodeBERT," Computers & Security, vol. 141, Art. no. 103994, Jul. 2024.
[46] NSA Center for Assured Software, "NIST SARD: Juliet Test Suite for C C plus plus," Version 1.3, Oct. 2017. doi: 10.5281/zenodo.4701387. Available: https://doi.org/10.5281/zenodo.4701387
[47] B. Steeves, "Big-Vul: A Large-Scale C/C++ Vulnerability Dataset," Hugging Face Datasets, 2022.

AUTHORS

Nin Ho Le Viet(First Author)is currently a Lecturer with the Faculty of Information Technology, Duy Tan University, Da Nang, Vietnam. He received the Engineering degree in Information Technology from the University of Science and Technology – the University of Danang in 2011, and the M.Sc. degree in Computer Science from Duy Tan University in 2015. His main research interests include information technology, software, artificial intelligence, and software security. He has co-authored several papers published in international journals indexed by ISI/Scopus and presented his work at national scientific conferences.

Tuan Nguyen Kim(Corresponding Author) received his Engineering degree in Electronics and Informatics from the University of Sciences – Hue University (1995), M.Sc. in IT from Hanoi University of Science and Technology (2003), and Ph.D. in Computer Science from Duy Tan University (2023). He is currently pursuing a second Ph.D. in Information Security at the Vietnam Academy of Cryptography Techniques. He serves as a lecturer at Phenikaa University and formerly held leadership roles at Hue University and Duy Tan University. His research focuses on post-quantum cryptography, secure protocols, and cybersecurity. He has published over 25 SCI/Scopus papers and 10 textbooks.

Nguyen Van Cuong is a researcher and lecturer in Computer Science at the School of Computing, Phenikaa University, Hanoi, Vietnam. He holds an M.Sc. degree in Computer Science, and his research focuses on source code vulnerability detection, cybersecurity, big language models, and data mining. His work centers on applying advanced machine learning and deep learning techniques to improve software security and analyze complex data-driven problems. He is actively involved in academic research and contributes to the development of secure and intelligent computing systems.

Chieu Ta Quang is a lecturer and researcher in the Department of Artificial Intelligence at Thuyloi University. He received his Ph.D. degrees in Computer Science from the University of Tours, France in 2015. His current research focuses on machine learning, deep learning, data analytics and big data. He has also published extensively in SCI/Scopus-indexed journals and international conferences and is the principal investigator or key member of multiple national-level research projects on digital transformation and intelligent systems in water resource management.