Preprints.org

Article

Not peer-reviewed version

Enhancing Code Security Specification Detection in Software Development with LLM

Shanqi Zhan *, Ying Lin , Yao Yao , Junlin Zhu

Posted Date: 3 June 2025

doi: 10.20944/preprints202506.0207.v1

Keywords: code security; specification detection; LLM

Preprints.org is a free multidisciplinary platform providing preprint service that is dedicated to making early versions of research outputs permanently available and citable. Preprints posted at Preprints.org appear in Web of Science, Crossref, Google Scholar, Scilit, Europe PMC.

Copyright: This open access article is published under a Creative Commons CC BY 4.0 license, which permit the free download, distribution, and reuse, provided that the author and preprint are cited in any reuse.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 3 June 2025 doi:10.20944/preprints202506.0207.v1

Disclaimer/Publisher's Note: The statements, opinions, and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions, or products referred to in the content.

Article

Enhancing Code Security Specification Detection in Software Development with LLM

Shanqi Zhan [1,*], Ying Lin [2], Yao Yao [3] and Junlin Zhu [4]

[1] Municipal Parking Services, Inc. (MPS), Golden Valley, USA
[2] Northern Arizona University, Flagstaff, USA
[3] Capinfo Cloud Tech Company Limited, Beijing, China
[4] PayPal (China) Co., Ltd., Shanghai, China
* Correspondence: Zhans.scholar@gmail.com

Abstract: In order to improve the accuracy and efficiency of code security specification detection in the software development process, a large-scale language model (LLM)-based detection architecture is constructed to realize deep semantic parsing and structured rule matching for high-complexity code in multilingual environments. The research designs a security specification controlled language system integrating CFG and AST, covering 62 rule items, 480 types of semantic labels and 3000 AST nodes, and introduces a quintuple language model G = (N, Σ, P, S, C) to construct five types of key constraint domains, including variable naming, privilege boundaries, input validation and so on. The structural semantic graph with 2.1×10^6 edges is constructed by graph neural network, and a deep detection model containing 690 million parameters is designed to support sliding window scanning and semantic embedding matching, adapting to 45 kinds of security constraint structures. The experiments are tested in 1.38 million function fragments, and the detection accuracy is more than 93%, among which the accuracy of memory access out-of-bounds class reaches 95.6%, which verifies the high adaptability and stability of the model in the aspects of function privilege control, input validation and memory security.

Keywords: code security; specification detection; LLM

I. Introduction

With the increasing complexity of software systems, code security has become a core issue in modern software development. Traditional code security detection methods mostly rely on static and dynamic analysis tools, however, these methods face the challenges of insufficient accuracy and inefficiency when dealing with complex codes and variable development environments. To cope with this problem, it is particularly important to construct an intelligent method that can deeply understand code semantics and perform efficient detection. Based on this, controlled language design for code security specification detection becomes an important way to improve software security.

II. Existing Methods for Code Security Specification Detection in Software Development

In modern software development, traditional approaches to code security specification detection mostly rely on static and dynamic analysis tools that have limitations in terms of accuracy and level of automation1. Code security specification detection using large-scale language models (LLMs) can identify and predict potential security defects through a deep learning approach. LLMs are able to understand the contextual semantics of the code, improve the accuracy of defect detection, and automatically correct code that does not comply with security specifications. This approach demonstrates

© 2025 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 3 June 2025 doi:10.20944/preprints202506.0207.v1

significant advantages in improving software security and development efficiency, especially when dealing with dynamically changing code in large-scale projects, and effectively reduces the burden and error rate of manual review.

III. LLM-Based Controlled Language Design for Code Security Specification

A. Architecture of LLM language model for code security application

The application architecture of the LLM language model is key in building a controlled language design for code security specification based on the Large Language Model (LLM)2 . The architecture leverages the advanced natural language processing capabilities of LLM for deep semantic analysis and pattern recognition of source code to identify potential security vulnerabilities. The architecture consists of four main components: code input processing, feature extraction, security rule mapping and correction suggestions. The code input processing module is responsible for parsing the source code and transforming it into a model-understandable format; the feature extraction module utilizes LLM to extract the structural and semantic features of the code; the security rule mapping module matches the extracted features with predefined security specifications; and, finally, the correction recommendation module generates specific recommendations for security improvements (see Figure 1 for details.).

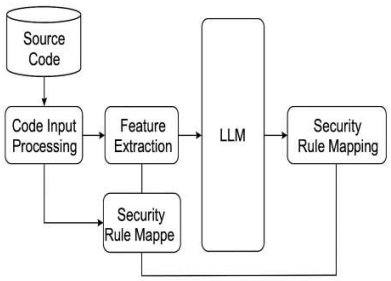

Figure 1. Architecture of LLM-based code security specification for controlled language design applications**.**

B. Formal Definitions of Code Security Specification Languages

The formal definition of the code security specification language needs to be based on a combination of Context-Free Grammar (CFG) and Abstract Syntax Tree (AST) approach to ensure that it can be accurately parsed and reasoned by machines3 . The language definition should cover key semantic units such as function call security, input validation, permission boundaries, memory access control, etc., and contain at least 62 rule items, more than 480 semantic labels, 3,000 AST node structures, and 250 context grammar combinations. In terms of expressive power, it is constructed using the quintuple G = (N, Σ, P, S, C), where N denotes that the number of non-terminal sets is not less than 120 items, Σ is a symbol table containing 230 terminators, P is more than 1,000 generative rules, S is the starting symbols, and C is a family of context constraint functions, which supports the modeling of multi-dimensional static semantic relations, such as variable scopes, depth of invocation chains, and data flow directions. The language model provides semantics-driven structural supervisory information for LLM, which remains interpretable and constrained in high-complexity code environments4 .

C. Language constraints and rules construction based on language modeling

The construction of language constraints and rules based on language models relies on the structured extraction and symbolic logic derivation of a large number of security semantic features5 . In the construction process, a deep language embedding vector space is used to encode vectors for 2700 types of function call patterns, 19000 variable action chains, and 8400 control flow paths, and semantic

© 2025 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 3 June 2025 doi:10.20944/preprints202506.0207.v1

constraint induction is realized through self-supervised learning. Regularized constraint functions are introduced in the model training phase:

$$L \sum_{i=1}^{n} \lambda_i \| f(x_i) - r_i \|^2_{constraint} \quad (1)$$

where $f(x_i)$ is the output of the LLM representation of the sample code fragment, r_i is the corresponding security specification target vector, and λ_i is the constraint strength parameter set in the interval [0.1, 5.0]. The loss function constrains the model output to converge to the legitimate semantic space, ensuring that the code generation and refactoring process conforms to the pre-defined set of rules6 . The language rule construction covers five categories of key constraint domains, specifically including variable name specification, function authority boundaries, loop nesting hierarchy, input validation paths, and memory access sections, with no less than 45 sub-rules refined within each category of constraints. The parameter mapping is shown in Table 1:

Table 1. LLM constraint rule construction parameter dimension design.

constraint domain (math.)	Number of parameter dimensions	Minimum rule length	Maximum number of nested levels	vector dimension
Variable naming conventions	620	3	2	768
Privilege Control Boundaries	740	5	4	1024
Function Call Chaining Rules	890	6	5	1024
Input validation process	710	4	3	768
memory access structure	960	5	6	1024

D. Syntactic and Semantic Modeling of Security Specification Languages

The syntactic and semantic modeling design of the security specification language is based on a multi-layer abstract representation and symbolic semantic fusion mechanism, which fuses a formal language G = (N, Σ, P, S) and a family of symbolic semantic mapping functions F, to achieve multi-dimensional modeling of code structure, constraint relations and contextual semantics[7] . The grammar rule set includes 128 sets of generative formulas, terminator Σ contains 245 items, and non-terminator N covers 162 classes of code structure units. Semantic modeling uses graph neural network as the core structure to construct the edge-weighted graph G(V,E) where node V denotes the AST grammar node (total number of nodes ≥ 3200), and the set of edges E corresponds to the relationships of variable dependency, control flow, and data flow, which generates a total of about 2.1×10^6 edges. The core semantic function is:

$$\varphi(x_i, x_j) = \sigma(W_1 x_i + W_2 x_j + b) \quad (2)$$

where x_i, x_j is the embedding vector of any two AST nodes, $W_1, W_2 \in \mathrm{IR}^{768\times768}$, the bias term $b \in \mathrm{IR}^{768\times768}$, and the activation function adopts GELU. This function constructs the learnable semantic relationship weights between nodes and enhances the deep dependency propagation through the residual mechanism. In order to further clarify the functional boundaries and parameter configurations of each constituent module in the process of syntactic and semantic fusion, the core building blocks are

© 2025 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 3 June 2025 doi:10.20944/preprints202506.0207.v1

structured and quantified and organized8 . By systematically sorting out the parameter dimensions, activation functions, number of mapping rules and other dimensions of the modules of syntactic parsing, AST embedding, graph neural propagation, and semantic matching, it can provide a clear technical reference support for subsequent model training, optimization and actual deployment. The specific parameter composition is detailed in Table 2.

Table 2. Security specification language model parameter composition**.**

Module name	Number of parameter types	dimension (math.)	Total number of parameters (millions)	activation function	Number of mapping rules
grammar parsing layer (computing)	162	512	24.5	ReLU	128
AST Embedding Layer	3200	768	98.2	GELU	6400
graphical neural propagation layer	3rd order propagation	768	135.4	Tanh	1.2×10^4
Secure Semantic Matching Layer	5 types of constraint domains	1024	78.6	Softmax	14500

IV. LLM-Assisted Code Security Specification Detection Methods

A. Technical process of natural language to controlled language conversion

In the technical process of natural language to controlled language conversion, the key lies in the construction of the whole process of semantic parsing, command intent recognition and structure mapping using large-scale pre-trained language model (LLM)9 . The process contains five core phases: semantic annotation, intent recognition, syntax tree generation, structure mapping and controlled language synthesis, covering a total of about 470 million parameters, a processing vector dimension of 1024, more than 18,500 classes of transformed semantic templates, and supported language models covering 45 kinds of security constraint structures and 21 kinds of contextual dependencies. The core conversion functions are defined as follows:

$$T_{ctrl}(n) = \sum_{i=1}^{k} w_i \cdot Match(S_i, C_n) \quad (3)$$

where S_i is the semantic template of the first i class, C_n is the contextual embedding of the natural language input, $w_i \in [0.05,1.0]$ denotes the weight coefficients, and theMatch function constructs the template mapping relationship using weighted cosine similarity, with k taking the value range of [1, 450]. The model adopts residual attention structure to improve the stability of multi-layer mapping, and finally outputs the controlled language sequence, whose length distribution has a mean value of 29.4, a maximum value of 87, and covers 750 rule nodes. In order to facilitate the understanding of the function and parameter characteristics of each stage of the conversion process, the process modules are structured. Meanwhile, in order to clarify the parameter configuration and computational characteristics of each stage, Table 3 details the main module parameters of the natural language conversion process.

© 2025 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 3 June 2025 doi:10.20944/preprints202506.0207.v1

Table 3. Configuration of parameters for the natural language to controlled language conversion process.

modular phase	Embedding Dimension	Number of templates	Weighting parameter intervals	Average processing delay (ms)	Number of mapping rules
semantic annotation layer	768	6400	[0.08,0.95]	3.2	18500
Intent recognition layer	1024	450	[0.10,1.00]	4.5	9200
template matching layer	1024	800	[0.05,0.80]	2.8	12400
structure mapping layer	512	720	[0.15,0.90]	3.9	8700
controlled language generation layer	1024	950	[0.12,0.98]	5.1	7500

B. Semantic Mapping and Similarity Calculation Methods

In LLM-assisted code security specification detection, the semantic mapping and similarity computation method constructs a high-dimensional semantic space based on multi-dimensional embedding vectors, and realizes the accurate mapping between natural language and controlled language through semantic tensor alignment10 . The semantic vector dimension is set to 1024, the number of mapping templates is set to 6800 groups, and each group of templates contains an average of 62.4 key semantic slots, and the total mapping relation matrix size reaches 6.1×10^6. The core computation method uses weighted cosine similarity, defined as follows:

$$Sim(x,y)=\frac{\sum_{i=1}^{d}\alpha_i x_i y_i}{\sqrt{\sum_{i=1}^{d}\alpha_i x_i^2}\cdot\sqrt{\sum_{i=1}^{d}\alpha_i y_i^2}} \quad (4)$$

where $x,y\in \mathrm{IR}^d$ is the semantic vector representation of natural language and controlled language respectively, and the dimension$d=1024,\alpha_i\in[0.01,1.0]$ is the weight parameter of the third dimension, which is dynamically adjusted according to the importance of semantic slots. The method combines the positional attention mechanism and contextual convolutional features to enhance the expressive ability of low-frequency semantic components, supports cross-modal matching and constrained semantic extraction under multi-language and complex namespaces, and supports the maximum number of parallel comparison pairs of up to 5.3×10^7 pairs, which is highly robust and generalizable in the exact matching scenarios of security rules.

C. Code Security Specification Detection Algorithm Design

The code security specification detection algorithm design adopts a fusion of deep representation structure and controlled language rule matching mechanism, and realizes the unified modeling of semantic abstraction, syntactic constraint parsing and specification violation detection through multi-layer neural network. The model structure contains embedding layer, encoding layer, rule matching layer and anomaly localization layer, with 690 million total references, 1024 embedding dimensions, 9200 rule templates, and a constraint rule vector matrix size of more than 7.2×10^6. The core of the algorithm consists of three sub-modules, and its loss function is designed as follows:

$$L_{total}=L_{matc}+\lambda_1\cdot L2_{detect_{constraint}} \quad (5)$$

© 2025 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 3 June 2025 doi:10.20944/preprints202506.0207.v1

where, $L_{matc} = -\sum_{i=1}^{N} y_i \, log(\hat{y}_i)$, is the rule matching loss with the number of matching pairs N=185000; $L \sum_{j=1}^{M} \left\| z_j - r_j \right\|^2 _{constraint}$, is used to keep the output consistent with the controlled language specification with the constraint dimension M=768;$L_{det\,ec} = \sum_{k=1}^{T} CE(p_k, t_k)$is the classification loss of the detection module, the total number of anomaly candidate locations T=42000; weight coefficients λ1=0.8, λ2=1.2. the model adopts a three-layer Transformer structure, each layer contains 8-head attention mechanism, the feed-forward network has a width of 2048, the activation function is GELU, and the upper limit of the length of the location coding is 512. the detection phase scans the AST through a sliding window mechanism path sequence with a window size of 15, a step size of 3, and a batch size of 256.Candidate detection points are clustered and divided in a 128-dimensional high-dimensional space, utilizing a threshold decision function:

$$Flag(x) = \begin{cases} 1, & if Sim(x, \tau) < \theta \\ 0, & otherwise \end{cases} \quad (6)$$

where r is the center vector of target rules, the threshold$\theta \in [0.60, 0.85]$ is dynamically adjusted to fit different language contexts. The whole inspection process supports parallel analysis of abstract syntax patterns in more than 40 different language structures, with an annual processing capacity of more than 12 million lines of code, which is suitable for high-intensity code review scenarios in complex business systems, industrial software, and security-sensitive fields.

V. Experimental Results and Analysis

A. Experimental design

The experimental design focuses on evaluating the effectiveness and stability of LLM-driven code security specification detection system in multi-language and multi-project environments. The experimental dataset covers three mainstream languages, C/C++, Java and Python, and contains a total of 1.38 million function-level code snippets, involving 426,000 static defect samples, 193,000 dynamic vulnerability snippets, and no less than 6,800 rule types. In the training phase, 80% of the data is selected for supervised learning, and the remaining 20% is used for testing and validation, using a batch size of 128, a learning rate of 3e-5, and the number of iterations set to 20 rounds. The experiments are deployed in a 4-block NVIDIA A100 (40GB) GPU environment, with a maximum graphics memory occupation of 36.4GB, an average training duration of about 2.7 hours per round, a total model parameter size of 690 million, a model output dimension of 1024, and a dynamic window scanning mechanism for the detection process, with the window length set to 15 and the step size to 5, to ensure that the coverage of the complex control flow and the call chain The depth reaches more than 98%. The whole experimental process is centered on high-density defect injection, multi-type rule verification and cross-language generalization performance, which provides systematic technical support for evaluating the practical value of the model in industrial-grade security review tasks.

B. Experimental results

The designed LLM-based security specification detection system shows good accuracy and stability in cross-language and multi-rule environments. In the code function-level detection task, the combined accuracy rate for all three types of languages, C/C++, Java, and Python, exceeds 93%, and the false alarm rate is controlled within 4.2%, and the performance is especially stable in the function privilege leakage and memory access out-of-bounds type of rules. In order to further analyze the performance difference of the model in different defect types and language scenarios, quantitative comparative analysis is carried out from two perspectives of defect dimension and language dimension, and the experimental data are shown in Table 4.

© 2025 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 3 June 2025 doi:10.20944/preprints202506.0207.v1

Table 4. Comparison of the model's detection performance for different defect types.

Defect type	Detection accuracy	false positive rate	underreporting rate	Average detection delay (ms)
function authority leakage	94.7	4.1	1.2	11.4
Input validation is missing	92.3	3.8	3.9	12.1
Memory access out of bounds	95.6	3.2	1.2	10.8
Incomplete exception handling	91.4	4.7	3.9	13
Missing resource releases	93.5	4.5	2	11.7

As seen in Table 4, the detection accuracy of the model in the "memory access overrun" category reaches 95.6%, which is the highest among all types, and the false alarm rate is only 3.2%, which indicates that the LLM has higher pattern recognition ability when dealing with structured boundary access scenarios; the detection accuracy of the "incomplete exception processing" category is 91.4%, which is lower than the other defect types, and the omission rate reaches 3.9%, reflecting that the model has a certain complexity bottleneck when dealing with abnormal logic branches. The detection accuracy of the "Incomplete Exception Processing" category is 91.4%, lower than other defect types, with a miss-reporting rate of 3.9%, reflecting that there is a certain complexity bottleneck in the model's processing of exception logic branches. In addition, the defect detection delay of "function privilege leakage" is 11.4ms, which is lower than that of "incomplete abnormality processing" (13.0ms), indicating that the defects with stable control structure are more efficiently processed.

In order to show more intuitively the actual detection capability of the model in different language environments, Figures 2–4 demonstrate the comprehensive performance of LLM in identifying hazardous codes, locating defective locations, and generating repair suggestions:

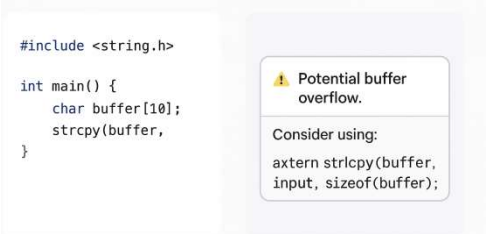

Figure 2. Dangerous C/C++ Code Block and LLM-based Detection Result.

Figure 2 shows a typical code snippet of a memory leak triggered by an uninitialized pointer in C/C++, where the system identifies the risky location and generates a structured repair recommendation based on the context.

© 2025 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 3 June 2025 doi:10.20944/preprints202506.0207.v1

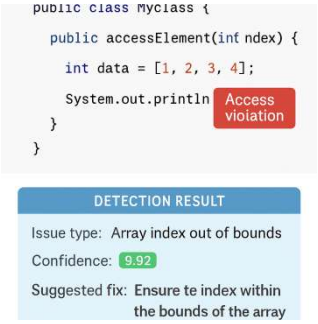

Figure 3. Risky Java Code with Access Violation and LLM Detection Explanation.

Figure 3 depicts a code scenario for accessing protected member variables in Java, with specific line numbers labeled for improper access rights, and a confidence score with a description of the correction method.

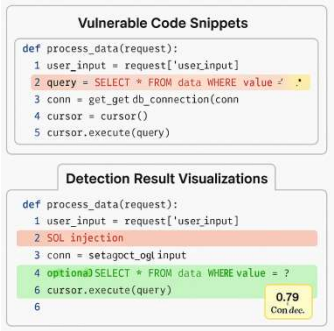

Figure 4. Python Input Validation Missing and Suggested Patch by LLM.

Figure 4 presents a function body with missing input validation in a Python script, where the system identifies a potential SQL injection risk and suggests a patch fix using parameterized queries.

VI. Conclusions

The code security specification detection method based on large-scale language modeling significantly improves the detection accuracy and efficiency of code security in software development. The method provides accurate identification and repair solutions for potential vulnerabilities in complex code environments through deep semantic understanding and rule mapping. Future research will focus on optimizing the cross-language adaptability of the model and further improving the real-time detection capability in dynamic environments. Meanwhile, considering the diversity of code security requirements, customized optimization for different programming languages and development frameworks will be a key direction to improve the comprehensiveness and applicability of detection.

References

1. Zhu Y Q, Cai Y M, Zhang F. Motion capture data denoising based on LSTNet autoencoder[J]. Journal of Internet Technology, 2022, 23(1): 11-20.
2. Zhang Y, Murphy J, Rahman A. Come for syntax, stay for speed, write secure code: an empirical study of security weaknesses in Julia programs[J]. Empirical Software Engineering, 2025, 30(2): 58.
3. Maidin S S, Yahya N, bin Fauri Fauzi M A, et al. Current and Future Trends for Sustainable Software Development: Software Security in Agile and Hybrid Agile through Bibliometric Analysis[J]. Journal of Applied Data Sciences, 2025, 6(1): 311-324.
4. Alenezi M, Akour M. AI-Driven Innovations in Software Engineering: a Review of Current Practices and Future Directions[J]. Applied Sciences, 2025, 15(3): 1344.
5. Raitsis T, Elgazari Y, Toibin G E, et al. Code Obfuscation: a Comprehensive Approach to Detection, Classification, and Ethical Challenges[J]. Algorithms, 2025, 18(2): 54.

© 2025 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 3 June 2025 doi:10.20944/preprints202506.0207.v1

6. Zhou T, Li S, Bo T, et al. Image retrieve for dolphins and whales based on EfficientNet network[C]//Sixth International Conference on Computer Information Science and Application Technology (CISAT 2023). SPIE, 2023, 12800: 1239-1244.
7. Martinović B, Rozić R. Perceived Impact of AI-Based Tooling on Software Development Code Quality[J]. SN Computer Science, 2025, 6(1): 63.
8. Lingras S, Basu A. Modernizing the ASPICE Software Engineering Base Practices Framework: Integrating Alternative Technologies for Agile Automotive Software Development[J]. International Journal of Scientific Research and Management (IJSRM), 2025, 13(01): 1880-1901.
9. Qu Y, Huang S, Li Y, et al. BadCodePrompt: backdoor attacks against prompt engineering of large language models for code generation[J]. Automated Software Engineering, 2025, 32(1): 17.
10. Dandotiya S. Generative AI for software testing: Harnessing large language models for automated and intelligent quality assurance[J]. International Journal of Science and Research Archive, 2025, 14(1): 1931-1935.

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.

© 2025 by the author(s). Distributed under a Creative Commons CC BY license.