

Automated Generation and Optimization Methods for Smart Contract Vulnerability Code Integrating Large Models and RAG Framework

Zuohu Chen^{1,*}, Zhenguo Peng¹, Yuntao Dou¹, Naicai He¹ and Wenxia Li¹

¹Gansu Tongxing Intelligent Technology Development Co., LTD., Gansu 730050, China

The application of big models to vulnerability generation for smart contracts is hampered by a lack of knowledge within the field, regularly producing difficulties such as logical anomalies, conceptual misunderstandings, and poor concealment within the generation results. In order to overcome the discrepancy between the poor quality of generated results and the important role played by big models within the field, the article proposes a retrieval-enhanced generation optimization strategy that combines a hybrid retrieval strategy and a strategy for a prompt and programmatic validation cycle constrained by pre-defined generation objectives. The results demonstrate that the strategy effectively enhances the authenticity level from 0.342 for the benchmark model CodeLlama (Code Large Language Model Meta AI) -13B to 0.867 and enhances the concealment score from 0.106 to 0.289, which contributes to improving the recognition capability level within the field by improving the quality and concealment level of generation, supplying a useful model for the proper use of big models within a relevant field. This research work presents a systematic optimization strategy for automatically optimizing the generation of data concerning smart contract security tests and adversarial examples. In addition to optimizing generation quality and concealment level, it functions as a relevant model for big models within a relevant field.

Keywords: Smart contract security; vulnerability code generation; large language model; search enhanced generation; automated testing

1. INTRODUCTION

The application of blockchain technology is no longer confined to cryptocurrency; it is now being used in finance, supply chain management, and the medical industry. Smart contracts, in essence, are the most important module on the virtual machines of blockchains. These contracts involve automated code that automatically executes critical opera-

tions such as asset transfers and permissions. Meanwhile, its security risks are also becoming increasingly prominent [1–2]. The security of smart contracts is directly linked to the presence of digital assets and has become a significant bottleneck in the increasingly expanding blockchain system [3]. While current traditional approaches such as static analysis testing, dynamic testing, and formal verification analysis tools have been extensively employed [4], these tools are heavily reliant on known pattern characteristics of vulnerabilities in smart contracts, and their detection power diminishes significantly in cases involving new or well-

*Corresponding Author: Email: ^aChengzHU198909@163.com, ^bpengzg1284@163.com, ^cDouyt0991@163.com, ^dHencai7324@163.com, ^eLiwenx0507@163.com

camouflaged logical-level vulnerabilities [5], making it difficult to address the increasing complexity of attack methods [6]. Thus, there is an extreme urgency to develop, via innovative approaches, technologically sophisticated security testing that is intelligent and proactive.

Existing approaches to detecting vulnerabilities in smart contracts tend to depend largely on patterns of known vulnerabilities, and do not provide an effective means of detecting unknown patterns of logic in smart contracts, or they tend to produce a large number of false positives. At the same time, although large language models have shown strong capabilities in terms of code generation [7], there are still significant flaws in actively generating exploitable vulnerabilities, and the generated results often only imitate vulnerability features on the surface syntax [8], lacking a complete chain of attack logic. This can easily confuse security concepts [9], ignore the contextual dependencies required for multi-contract interactions, and make it difficult to grasp the concealment and evasion strategies required to combat real attacks [10]. The generation of results that resemble but are not genuine is essentially due to the lack of in-depth understanding of smart contract security mechanisms in the model, especially the lack of precise, structured, and inferential domain knowledge support [11–12]. Therefore, in order to fully utilize the potential of large models in security testing, it is necessary to systematically optimize their generation process to enable them to produce high-quality vulnerability samples that are real, hidden, and exploitable in practice, thus breaking through the current bottleneck of adversarial testing and security tool enhancement applications.

To address this research gap, this study systematically reviews the application of smart contract vulnerability detection, large models in the field of security, and the cutting-edge progress of Retrieval-Augmented Generation (RAG) technology. Existing research has achieved preliminary results in RAG-assisted code generation, but a structured knowledge system has yet to be constructed specifically for smart contract vulnerabilities [13], and there is a lack of systematic modeling of vulnerability principles, context dependencies, and attack chain logic [14]. More importantly, the current works lack a closed-loop verification mechanism to ensure the authenticity of generated samples [15]. Collectively, these shortcomings reveal that the key to achieving high-quality vulnerability generation lies in designing an end-to-end optimization method that comprises knowledge, retrieval, and verification. This study addresses three research gaps: 1) the lack of a structured knowledge base for vulnerability generation [16]; 2) the lack of a hybrid retrieval strategy adapted to vulnerability semantics [17]; and 3) the lack of iterative optimization mechanisms for generation validation feedback [18]. Collectively, these issues constrain the practical value of large models in smart contract security testing, and there is an urgent need for a new method that integrates domain knowledge and generative intelligence.

In response to the above challenges, this study introduces a smart contract vulnerability automatic generation method that integrates the RAG framework, and combines the mixed retrieval mechanism of CodeBERT (Code Bidirectional

Encoder Representation from Transformers) and BM25 (Best Matching 25) to accurately guide large models to generate code that conforms to attack principles. At the same time, this study designs a dynamic prompt template and a closed-loop verification process of “generation compilation detection feedback” to ensure the authenticity and exploitability of vulnerabilities. The experimental results show that the vulnerability code generated by this method reaches 0.867 in terms of authenticity, which is a significant improvement compared to the CodeLlama-13B baseline (0.342), and the concealment is enhanced. The generated samples can enhance the existing Slither model. The proposed approach achieves a paradigm shift from “passive detection” to “active generation”; its main contribution is the introduction of a technical path that can systematically optimize the performance of large-scale model generation, providing high-quality adversarial samples and a scalable technical framework for the security testing of smart contracts.

2. RAG-BASED SMART CONTRACT VULNERABILITY CODE GENERATION AND OPTIMIZATION METHOD

2.1 Formal Definition of Problems

The problem of generating vulnerability code for smart contracts can be formally defined as a conditional probability distribution optimization problem. Given a set of target vulnerability types $V = \{v_1, v_2, \dots, v_m\}$, where each v_i corresponds to a specific vulnerability type in the SWC (Smart Contract Weakness Classification) standard, as well as a contract function description F , it is necessary to generate smart contract code C that meets the following conditions:

Vulnerability authenticity: C must contain at least one real vulnerability belonging to V , and the vulnerability can actually be exploited.

Code quality: C should have good engineering properties, including correct syntax, successful compilation, and basic compliance with smart contract coding standards to ensure that it can be used as an effective testing and analysis object.

Concealment: Vulnerabilities in C should have a certain degree of concealment to avoid being easily identified by mainstream detection tools.

Diversity: For the same V and F , multiple valid vulnerability codes $\{C_1, C_2, \dots, C_k\}$ with different structures and implementation methods should be generated.

Novelty: The generated vulnerability code should be able to partially surpass known simple patterns and provide vulnerability implementation methods.

From a mathematical perspective, the objective of this study is to optimize the generation distribution $P(C|V, F, K)$, where K is the set of domain knowledge. The traditional generation of large models can be represented as $P_\theta(C|V, F)$, but its drawback is that the knowledge encoded by the parameter θ cannot accurately cover the complex concepts in the field of smart contract security. After introducing the RAG framework, this article dynamically

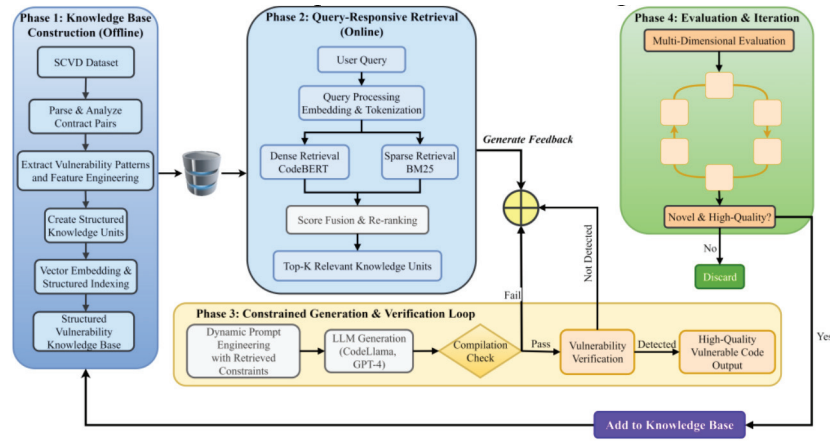


Figure 1 Smart contract vulnerability code generation and optimization method with rag framework fusion.

introduces external knowledge $K R(V, F)$ through a retrieval mechanism. R is the retrieval function, which optimizes the generated distribution to $P_{\theta}(C|V, F, K)$, thereby improving the probability of meeting the above conditions. To quantify the quality of vulnerability generation, define a core evaluation function:

$$Q_{core}(C) = \lambda_1 \cdot A(C) + \lambda_2 \cdot D(C) + \lambda_3 \cdot S(C) + \lambda_4 \cdot Q(C) + \lambda_5 \cdot Nov(C) \quad (1)$$

In formula (1), $A(C)$ represents the authenticity score, which measures the exploitability of vulnerabilities; $D(C)$ is the diversity score, which represents the degree of difference between the generated samples of the same type; $S(C)$ is the concealment score, the complement of mainstream tool detection rate; $Q(C)$ is the code quality score, and $Nov(C)$ is the novelty score; The weight coefficients are $\lambda_1, \lambda_2, \lambda_3, \lambda_4, \lambda_5$.

2.2 Overall Framework of the Method

The smart contract vulnerability code generation method introduced in this study, which integrates the RAG framework, consists of four core stages forming a complete closed loop of “knowledge construction retrieval enhancement generation verification iterative optimization”, as shown in Figure 1.

Figure 1 shows the four-stage workflow of automated generation of vulnerability-aware smart contracts, which is divided into two modules: offline knowledge construction and online generation iteration, with both systematic and closed-loop optimization characteristics.

Phase 1: Construction of a structured vulnerability knowledge base, converting raw smart contract vulnerability data into structured and retrievable knowledge units. Based on the SCVD (Smart Contract Vulnerability Dataset) dataset, the focus is on handling “vulnerability repair” contract pairs. Through AST parsing and code differentiation techniques, the code change points involved in the repair are accurately located. Each vulnerability sample is broken down into structured entities containing vulnerability types, vulnerability and fix code snippets, natural language principle descriptions, and contextual information. This study uses CodeBERT to

vectorize code and text, and stores them in a vector database. At the same time, it constructs negative samples using secure contracts to enhance the knowledge base’s ability to distinguish between secure and insecure patterns.

Phase 2: The query-response retrieval phase can build a hybrid retrieval engine for smart contract vulnerability generation. After users introduce vulnerability generation requirements in natural language, the system converts them into semantic vectors and extracts keywords. Candidate knowledge units are obtained through CodeBERT dense retrieval [19] and BM25 sparse retrieval [20]. In this study, the results are ranked by means of weighted fusion and the Top-5 most relevant structured vulnerability-knowledge results are returned, providing accurate and context aligned domain knowledge support for subsequent generation.

Phase 3: Constraint generation and programmatic verification closed loop: Based on the search results, the system uses dynamic prompt engineering to embed vulnerability patterns, repair comparisons, and principle descriptions into the large model prompts, guiding CodeLlama[21] or GPT (Generative Pre-trained Transformer) –4 [22] to generate code that conforms to the attack logic. The generated results immediately enter the automated verification loop: firstly, the Solidity compiler checks the syntax to detect the existence of target vulnerabilities; finally, this study simulated and deployed the attack path on the test network to verify its usability. If the verification fails, the system converts the tool feedback into new constraints and iterates up to three times to optimize the generated results, ensuring that the vulnerability is real and can be triggered.

Phase 4: Multidimensional evaluation and system iteration: To systematically measure the generation effect, five evaluation dimensions are defined in this stage: authenticity (availability), diversity (sample differences), stealthiness (bypassing detection tools), code quality (readability and standardization), and novelty (surpassing known patterns), and a comprehensive score is calculated. This study verified the effectiveness of each component of RAG by comparing it with a pure large model baseline and ablation experiments. At the same time, high-quality generated samples that have been manually reviewed and approved can be fed back to the knowledge base, achieving continuous expansion of the knowledge base and self-enhancement of system capabilities,

forming a virtuous cycle of “generation, verification, and learning”.

In addition to the basic assessment of generation quality, this study also designed two high-order capability verification experiments to comprehensively evaluate the practical value of the proposed method in real security scenarios.

- (1) This study reverse-injected the high-quality vulnerability samples generated into the training process of the mainstream detection tool, Slither, verifying its effectiveness in improving the tool’s ability to identify unknown or highly covert vulnerabilities, forming a positive cycle of “generation, enhancement, and re detection”.
- (2) This study tested the collaborative generation capability of the system when user requests involve multiple types of vulnerabilities, evaluated its ability to construct composite vulnerabilities with reasonable interaction logic, and simulated increasingly complex advanced persistent threat scenarios in the real world.

2.3 Construction of Structured Vulnerability Knowledge Base

2.3.1 Multi-Source Dataset Collection and Preprocessing

This study selected SCVD (Smart Contract Vulnerability Dataset) as the core data source, which consists of subsets of Detection Dataset and Mitigation Dataset, specifically designed for research on smart contract vulnerability detection and repair.

The Detection Dataset contains the source code and binary labels (security/vulnerability) of 35228 Ethereum smart contracts, covering mainstream scenarios such as tokens, DeFi (Decentralized Finance) protocols, DAO (Decentralized Autonomous Organization) governance, and more. Each contract is validated for tag consistency by at least three static analysis tools, ensuring good data balance.

The Mitigation Dataset contains 10448 pairs of carefully labeled “vulnerability contract repair contract” pairs, covering 28 types of SWC standard vulnerabilities, including high-frequency vulnerabilities such as re-entry attacks (SWC-107), integer overflow (SWC-101), access control defects (SWC-103), business logic errors (SWC-105), timestamp dependencies (SWC-116), and oracle manipulation risks (SWC-157). Each contract not only contains source code, but also provides detailed vulnerability location tags and repair instructions.

2.3.2 Vulnerability Knowledge Extraction and Annotation

Knowledge extraction is a key step in transforming original contract pairs into structured knowledge units, consisting of four sub processes:

- (1) Contract parsing and AST (Abstract Syntax Tree) construction: This study used the Solidity compiler to

compile vulnerability contracts and repair contracts into AST [23], and obtained contract structure information and code logic by traversing the AST. AST parsing can accurately identify the syntax structure of code and avoid ambiguity in text analysis.

- (2) Function level alignment and code difference extraction: To accurately identify the key modification points for vulnerability fixes, this study paired the functions of vulnerability contracts and repair contracts. It adopted precise matching priority based on function signature pairing, used similarity matching for function signature changes, and used token-based cosine similarity to assist matching. The formula for calculating the similarity between functions $f1$ and $f2$ is:

$$\text{sim}(f1, f2) = \frac{v_{T1} \cdot v_{T2}}{|v_{T1}| |v_{T2}|} \quad (2)$$

where $T1$ and $T2$ are the token sequences of the function code, and vT is the TF-IDF (Term Frequency Across Document Frequency) vector of the token sequence.

After pairing, use the AST based differential algorithm to extract code modifications. Compared with traditional text differentiation tools, AST differentiation can recognize semantic level changes rather than just text changes. The formula for calculating the degree of difference is:

$$\text{diff}(A, B) = 1 - \frac{2 \times l}{|A| + |B|} \quad (3)$$

where A and B are the AST node sequences of two functions, and l is the longest common subsequence length. The higher the degree of difference, the greater the modification of the function logic.

- (3) Vulnerability pattern abstraction: Associate extracted code differences with vulnerability types to form reusable vulnerability patterns. The abstraction process follows the principle of “minimum necessary modification”, which identifies the smallest code unit that triggers the vulnerability.

Vulnerability triggering conditions: such as “the contract allows external calls and the state is not locked before the call”;

Key code positions: such as “relative order of external call statements and state update statements”;

Fix method: such as “update state variables first and then execute external calls” or “use reload lock modifier”.

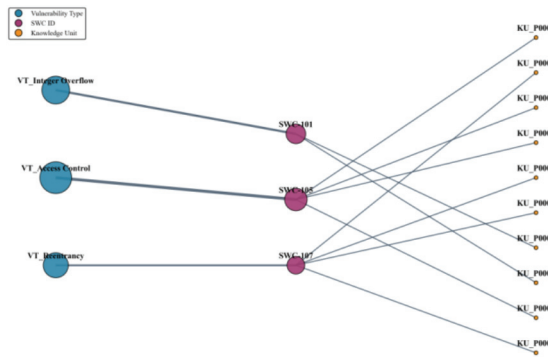
- (4) Feature engineering: structured features of vulnerability patterns are extracted to enhance the retrieval capability of knowledge units.

Code features: function name, state variable name, key operator;

Semantic features: natural language description of vulnerability principles;

Table 1 Structured knowledge unit contains fields.

Field	Description	Data Type
vulnerability_id	Unique identifier for the vulnerability	String
vulnerability_type	Type/category of vulnerability	String
contract_name	Name of the smart contract	String
function_signature	Signature of the vulnerable function	String
compiler_version	Solidity compiler version used	String
vulnerable_code	Code snippet containing the vulnerability	String
fixed_code	Corrected code snippet (fixed version)	String
diff_description	Description of the vulnerability root cause and fix	String
code_vector	Vector embedding of vulnerable code	Array[Float]
desc_vector	Vector embedding of difference description	Array[Float]
features	Structured feature set	Object
key_functions	Key functions involved in vulnerability	Array[String]
state_variables	State variables involved in vulnerability	Array[String]
access_modifier	Access modifier of the function	String
vulnerability_keywords	Keywords describing the vulnerability	Array[String]
metadata	Additional metadata (optional)	Object
source	Source of knowledge unit	String
expert_verified	Whether verified by security expert	Boolean
creation_date	Date when knowledge unit was created	String
severity	Severity level of vulnerability	String

**Figure 2** Knowledge graph visualization.

Context features: compiler version, contract type, permission modifier.

These features are used to construct a hybrid retrieval index and improve the retrieval accuracy of the RAG system.

2.3.3 Dimensionality Representation and Storage Strategy

(1) Definition of Structured Knowledge Units

Each knowledge unit is a JSON (JavaScript Object Notation) object containing complete vulnerability information, formatted as shown in Table 1.

The structured knowledge units for smart contract vulnerabilities defined in Table 1 integrate three key dimensions: executable code (vulnerable_code, fixed_code), interpretable principles (diff_description), and searchable features (code_vector, desc_vector, features and their metadata), supporting multimodal knowledge representation for deep understanding and precise matching. Figure 2 shows the first

ten extracted knowledge units and displays the topological relationship of the structured knowledge base in a hierarchical visualization form, clearly showing the three-layer structure of VT (Vulnerability Type), SWC standard, and specific KU (Knowledge Unit).

Figure 2 uses a three-layer topology to organize 10 knowledge units. These 10 units cover three main vulnerability types (Reentrancy, Integer Overflow, and Access Control) and their corresponding three SWC standards (SWC-107, SWC-101, and SWC-105). The thickness of the edges indicates the strength of the association; the Reentrancy and Access Control types have a larger number of knowledge units. The graph structure provides the foundation for subsequent RAG retrieval, allowing the system to quickly locate relevant knowledge along these pre-built relational paths during retrieval.

(2) Knowledge Vectorization

The pre-trained code model CodeBERT[24] is used to vectorize knowledge units. CodeBERT is pre-trained on code corpora of 21 programming languages, effectively capturing code syntax and semantic features. The vectorization process

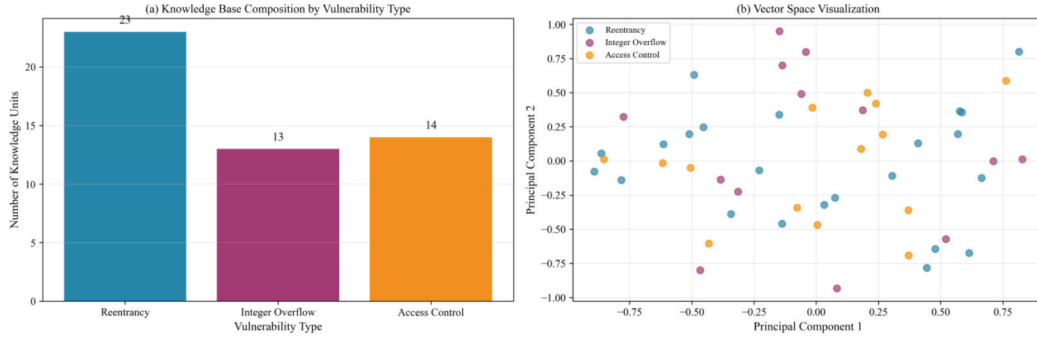


Figure 3 Knowledge base structure diagram. (a) Knowledge base composition analysis. (b) Vector space visualization

is as follows: **Input Processing:** The vulnerability code fragment and the difference description are concatenated into strings, and special markers ([CLS], [SEP]) are added; **Model Encoding:** The output vector at the [CLS] position is obtained through the CodeBERT model and used as the representation of the entire sequence; **Vector Storage:** The code vector (code_vector) and the description vector (desc_vector) are stored in the vector database Chroma, which supports efficient approximate nearest neighbor search.

(3) Storage Architecture

The knowledge base has a hybrid storage architecture. The vector database Chroma stores code_vector and desc_vector, supporting intensive retrieval; the relational database SQLite (Structured Query Language Lite) stores structured fields, supporting SQL (Structured Query Language) queries and keyword matching; and the file system stores the complete contract source code for in-depth analysis and manual review. Figure 3 shows 50 representative knowledge units extracted from the complete knowledge base for visualization analysis.

Figure 3(a) shows that Reentrancy accounts for the highest proportion (23 units) among the three vulnerability types, consistent with the high incidence of this type of vulnerability in smart contracts. Figure 3(b) uses PCA (Principal Component Analysis) to reduce the dimensionality of the feature vectors. The three types of vulnerabilities (Reentrancy, Access Control, and Integer Overflow) exhibit high overall separation in the embedding space, and do not form a tight cluster structure within themselves. This indicates that although the model can distinguish between different types of vulnerabilities, there are still significant internal differences in code semantics or structure among similar vulnerabilities, and their distribution in the embedding space is relatively scattered.

2.4 Domain Knowledge Driven Hybrid Retrieval Strategy

The hybrid retrieval strategy designed in this study combines the advantages of both dense and sparse retrieval to achieve complementary efficiency.

(1) Intensive search

A similarity search based on CodeBERT vectors is conducted to capture semantic associations between queries and knowledge units [25]. Convert user generated requests into vectors. Since the request is in natural language,

first, it needs to be associated with the “vulnerability code generation” task. The prompt template is “Convert the following vulnerability generation request into a retrieval vector: {user_query}”. Use the same CodeBERT model as the knowledge unit to encode the query and generate the vector q_d . Calculate the cosine similarity between q_d and all code-vectors and desc-vectors in the knowledge base:

$$\text{similarity}(q_d, v) = \frac{q_d \cdot v}{|q_d| |v|} \quad (4)$$

This study combines the similarity scores of the code vector and the description vector in formula (4), and takes the Top-5 knowledge units as the search result.

(2) Sparse Search

Keyword matching is done based on the BM25 algorithm, utilizing structured features to improve retrieval accuracy. The core idea of BM25 is to calculate the weight of query terms in the document [26] with the formula:

$$BM25(D, Q) = \sum_{q \in Q} IDF(q) \cdot \frac{f(q, D) \cdot (k_1 + 1)}{f(q, D) + k_1 \cdot \left(1 - b + b \cdot \frac{|D|}{avgdl}\right)} \quad (5)$$

where D is the document, Q is the set of query terms, $f(q, D)$ is the frequency of word q in D , $|D|$ is the document length, $avgdl$ is the average document length, k_1 , b are adjustment parameters. In this system, the sparse retrieval ‘document’ is the features field of the knowledge unit, and the ‘query term’ is extracted from the user request.

(3) Hybrid fusion strategy

Using a weighted score fusion method to merge two types of search results:

$$\text{score}_{final}(K) = \alpha \cdot \text{score}_{dense}(K) + (1 - \alpha) \cdot \text{score}_{sparse}(K) \quad (6)$$

where K is the knowledge unit and α is the weight parameter. After fusion, the knowledge units are sorted by criterion score_{final} . The Top-K knowledge units obtained through the preliminary retrieval may have redundancy or insufficient relevance, and further optimization of the sorting is needed. The similarity between knowledge units is calculated, highly similar units are filtered, and the diversity is preserved. In the search results, if there are too many of a certain type

of vulnerability, the weight of other related types should be appropriately increased to ensure the diversity of the generation. Scores are adjusted based on the contextual information in user requests. After reordering, the system returns the final Top-5 knowledge units as generating constraints.

2.5 Constraint Generation and Verification of Vulnerability Code

2.5.1 Dynamic Prompt Template and Constraint Injection

Dynamic prompt engineering is a key step in transforming retrieval knowledge into generation constraints that can be understood by large models. The prompt template design consists of four parts.

The task instructions clearly define the generation target, including vulnerability types, contract functions, and special requirements.

Retrieval knowledge injection is the structured embedding of key information from the Top-5 knowledge units into prompts. Each knowledge unit includes a brief description of the vulnerability principle, a comparison between vulnerability code and repair code, and vulnerability triggering conditions.

Generative constraints specify the conditions that code must meet, including syntactical constraints, functional constraints, and vulnerability constraints.

Output format specifies the structure of the generated results for subsequent automated verification: “Output format: Vulnerability contract code: Complete Solidity code, including detailed comments. The attack contract code is used to verify vulnerabilities in the attack contract. Vulnerability description briefly explains the location of the vulnerability and the triggering conditions”.

2.5.2 “Generate compile test” single validation process

To ensure the authenticity and exploitability of generated vulnerabilities, this study designed a four-step iterative optimization loop of “generation compilation detection feedback”, using the Solidity compiler to compile the generated contract and check for syntactical errors and version compatibility. If compilation fails, error information can be extracted as part of the feedback prompt.

This study used Slither and Mythril, two mainstream static analysis tools, for vulnerability detection. Slither [27] detects vulnerabilities in code data flow and control flow, such as reentrancy, integer overflow, and access control flaws; Mythril[28] combines symbolic execution to detect potential state tampering and asset theft paths.

There are three types of detection results: successful detection of target vulnerability that clearly states the type of target vulnerability; target vulnerability is not detected in that it does not display the target vulnerability but might display other ones; false positive that states the target vulnerability even if a manual analysis confirms that it has not been installed in the system by. The detection result in the tool can be expressed as a natural language statement.

Vulnerability contract and attack contract of attack simulation are deployed for the local test environment, and attack path is performed to test the exploitability of vulnerability. The attack simulation result is shown as follows: Success: Attack contract can acquire unauthorized assets based on vulnerability; Failure: The attack process invokes rollback or fails to reach expected result; Partial Success: It triggers vulnerability but fails to acquire unauthorized assets. Attack simulation is recognized as the most direct and strict criterion used to test the authenticity of vulnerability.

Feedback prompts integrate compilation errors, detection reports, and attack results into natural language feedback.

2.6 Iterative Optimization and System Self-Enhancement Mechanism

2.6.1 Definition of Evaluation Index System

A (authenticity): Evaluate whether the generated vulnerability code truly reflects the target vulnerability pattern and is exploitable in a real-world environment.

$$A = \eta_1 \cdot \frac{N_v}{N} + \eta_2 \cdot \frac{1}{N} \sum_{i=1}^N S_i + \eta_3 \cdot \frac{1}{N} \sum_{i=1}^N E_i \quad (7)$$

where N is the total number of samples, N_v is the number of samples detected as vulnerabilities by security tools, S_i is the semantic similarity between the sample and the target vulnerability, E_i is the exploitability score of the sample on the testnet, and η_1, η_2, η_3 are weighting coefficients ($\eta_1 = 0.5, \eta_2 = 0.3, \eta_3 = 0.2$).

D (Diversity) assesses the richness of variants of the generated vulnerability, including different implementations within the same type (intra-class diversity) and the distinguishability between different types (inter-class diversity). The calculation formula is:

$$D_{intra} = \frac{1}{T} \sum_{t=1}^T \left[\frac{2}{n_t(n_t - 1)} \sum_{i=1}^{n_t-1} \sum_{j=i+1}^{n_t} \left(1 - \cos[f_0(v_i^t, v_j^t)] \right) \right] \quad (8)$$

$$D_{inter} = \frac{2}{T(T-1)} \sum_{t=1}^{T-1} \sum_{k=t+1}^T \left(1 - \cos[f_0(\hat{v}^t, \hat{v}^k)] \right) \quad (9)$$

$$D = \lambda \cdot D_{intra} + (1 - \lambda) \cdot D_{inter} \quad (10)$$

where T represents the total number of vulnerability types, n_t represents the number of samples of type t , v_i^t represents the vector representation of the samples, and $\hat{v}^t$ represents the average vector of type t . In formula (10), $\lambda = 0.6$ is the intra-class diversity weight.

S (Stealthiness): Evaluates the ability of generated vulnerabilities to evade mainstream security detection tools. The calculation formula is as follows:

$$S = 1 - \frac{1}{M} \sum_{m=1}^M w_m \cdot R_m \quad (11)$$

where M represents the number of detection tools, R_m represents the detection rate of tool m , and w_m represents the weight of tool m . In this study, the detection tools are Slither and Mythril, $M = 2$, and $w_m = 0.5$.

Q (Code Quality): Evaluates the basic quality of the generated code, ensuring that the code is usable and conforms to specifications. The calculation formula is:

$$Q = \omega_1 \cdot C_a + \omega_2 \cdot C_s + \omega_3 \cdot C_{st} \quad (12)$$

where C_a , C_s , and C_{st} represent compilation pass rate, syntax accuracy, and encoding standard compliance rate, respectively, with corresponding weights of $\omega_1 = 0.4$, $\omega_2 = 0.3$, $\omega_3 = 0.3$.

Nov (Novelty): Evaluate the novelty of generated vulnerabilities relative to existing patterns in the knowledge base, calculated using the following formula:

$$Nov = \frac{1}{N} \sum_{i=1}^N \min_{j \in D_{train}} \text{ditDistance}(c_i, d_j) \quad (13)$$

where c_i is the generated vulnerability code, d_j is the code sample in the training knowledge base, and EditDistance is the function that measures the minimum editing distance between the two.

Comprehensive performance: comprehensively evaluate the overall performance of the generation system, balancing the performance of various dimensions. The rating formula is:

$$Overall = \sum_{m \in M} \beta_m \cdot S_m \quad (14)$$

where $M = \{A, D, S, Q, Nov\}$ is the set of indicators mentioned above, S_m is the original score of indicator m , and $\beta_m = [0.25, 0.20, 0.20, 0.15, 0.20]$ is the weight of each indicator.

Coverage: The percentage of queries in a set of test queries that the retrieval system is able to return at least one relevant knowledge unit. The formula is expressed as:

$$\text{Coverage} = \frac{\text{Number of queries with at least one relevant result in top-}K}{\text{Total number of test queries}} \times 100\% \quad (15)$$

where $K = 5$, and the system returns a Top-5 result.

Top-5 Accuracy: The proportion of queries in which at least one relevant document is included in the top 5 results of the test query. When generating vulnerabilities in smart contracts, the retrieval results can be injected into the large model as contextual knowledge prompts. Due to the limited length of the prompt, only the first 5 search results are considered for practical applications. The Top-5 accuracy directly indicates the effectiveness of the retrieval system in practical usage scenarios.

Average retrieval time: end-to-end retrieval delay, measured in milliseconds.

A success rate indicator is used for specific vulnerability generation tasks. The success rate of generation is indicated by the proportion of samples that ultimately pass the complete programmatic validation loop. The verification loop includes three necessary conditions: (1) the code can be successfully compiled by the Solidity compiler; (2) the target

vulnerability type is identified by at least one professional detection tool, or confirmed by manual verification to have a valid attack path; (3) the executed attack simulation has been successfully executed and achieved unauthorized asset transfer in the local test network. The calculation formula is:

$$GSR = \frac{1}{N_{total}} \sum_{i=1}^{N_{total}} I \quad (C_i \text{ passes compilation} \wedge \text{vulnerability validated} \wedge \text{exploit succeeds}) \quad (16)$$

where N_{total} is the total number of generated samples, and $I(\cdot)$ is the indicator function. This indicator can accurately express the end-to-end task completion ability of the system in real security research and testing environments, and can be used for testing the feasibility of generating complex vulnerabilities under multiple constraints.

2.6.2 Knowledge Base Self-Enhancement and Iterative Update Strategy

In order for the system to constantly evolve and self-improve, this study proposes a dynamic iterative update in the knowledge base where high-quality synthetic samples are fed back into the knowledge base so as to achieve an optimization process of “generation verification learning” to enable the system to accumulate new vulnerability patterns.

The iterative approach begins with the rigorous selection of the synthesized samples based on quality, ensuring that only the best-performing ones that excel in terms of key indicators including authenticity, diversity, and stealth succeed in this phase, in addition to having a high level of novelty. Later on, manual analysis by the experts would enable the highlighting of accuracy in terms of the vulnerability principle, self-triggering abilities in relation to the code, and the security consequences.

The qualified samples can then be applied to generate structured units of knowledge according to specified details such as vulnerability codes, fix plans, descriptions of differences, as well as other related metadata, distinctly different from the original sources of knowledge. Finally, with the application of vector similarity deduplication approaches to guarantee the uniqueness of the novel knowledge, the unit of knowledge is formally added to the database with source information for generation, thereby closing the loop on updates to the unit of knowledge.

3. EXPERIMENT AND RESULT ANALYSIS

3.1 Experimental Setup

3.1.1 Dataset and Baseline Model

Based on the SCVD experiment, the initial knowledge database was established using the Mitigation Dataset’s randomly chosen 8358 contracts for vulnerability fixes. The validation of the matches between the generated vulnerabilities and actual vulnerability patterns was conducted on the Mitigation Dataset’s 2090 contracts. The validation of the vulnerability detectability and the false positive ratio

Table 2 Technology stack details.

Module	Technology Choice	Role / Function
Data Parsing	solc (Solidity Compiler), AST Parser	Parse smart contracts into Abstract Syntax Trees to extract structural and semantic code elements.
Knowledge Embedding	CodeBERT	Generate dense vector embeddings for both vulnerable code snippets and natural language vulnerability descriptions.
Vector Storage	Chroma	Store and index vector embeddings to enable efficient approximate nearest neighbor search for dense retrieval.
Retrieval Engine	CodeBERT, BM25	Implement a hybrid retrieval system combining dense (vector-based) and sparse (keyword-based) search; LangChain orchestrates the retrieval pipeline and result fusion.
LLM	LangChain, GPT-4	Generate syntactically and semantically plausible smart contract code conditioned on retrieved knowledge and constraints.
Verification Tools	Slither, Mythril, Hardhat	Perform multi-stage validation: Slither, Mythril (symbolic execution for state-based exploits), Hardhat (local testnet deployment and attack simulation).
Experiment Management	ML(Machine Learning)flow	Track and manage experiments, including hyperparameters, evaluation metrics, model versions, and generated artifacts.

was carried out on the Detection Dataset’s randomly-chosen 5000 contracts (concerning 2500 vulnerability contracts and 2500 security contracts). All dataset partitioning ensures the uniform distribution of vulnerability types through stratified sampling, avoiding excessive or insufficient representation of certain types of vulnerabilities.

To verify the effectiveness of this method, the following comparison methods were established:

(1) Pure large model generation:

Baseline 1a: CodeLlama-13B [29], a code specific large model published by Meta, performs excellently in code generation tasks;

Baseline 1b: GPT-4 [30] is a general-purpose large model released by OpenAI, possessing powerful natural language understanding and code generation capabilities. It provides only natural language descriptions and does not use any retrieval knowledge.

(2) Traditional RAG Generation:

Baseline 2 uses the general RAG framework, LangChain [31] + GPT-4. The knowledge base is the original vulnerability contract text, and the retrieval strategy uses only intensive retrieval.

Ablation experiments were conducted to quantitatively analyze the contribution of each component to the final generation quality by progressively removing or replacing key components in the RAG framework (retrieval tool, knowledge base, constraint hints, and procedural verification loop).

To evaluate the adaptability of different retrieval strategies to the structured vulnerability knowledge base, three methods were compared on a knowledge base containing 10,448 structured vulnerability knowledge units: pure sparse retrieval (BM25), pure dense retrieval (CodeBERT), and hybrid retrieval (weighted fusion, $\alpha = 0.7$).

Furthermore, this experiment draws on the dimensionality reduction-based anomaly detection framework proposed by Sipola et al. (2012) for the design of evaluation metrics and feature analysis, and references the resilient cybersecurity assessment standards of Zipperle et al. (2023) to enhance

the comprehensiveness of the experimental methodology in feature space optimization and security validation [32–33].

3.1.2 Evaluation Metrics and Experimental Environment

The hardware configuration for this study was an Intel Xeon Gold 6248 CPU, two NVIDIA A100 80GB GPUs, and 128GB of RAM. The software environment consisted of Ubuntu 20.04 LTS, Python 3.9, Solidity 0.8.17, and Hardhat 2.12.4. The experimental technology stack is shown in Table 2.

Table 2 shows the technology stack of this method. The solc and AST parser transform the original contract into a structured syntax tree and extract fine-grained code semantics from it. The vulnerability code and its natural language description are vectorized by CodeBERT and stored in the Chroma vector database to support efficient semantic retrieval. In the generation stage, the system adopts the LangChain framework to integrate BM25 keyword retrieval and vector similarity search to achieve hybrid retrieval and dynamically construct hints containing domain knowledge to drive GPT-4 to generate contract code that meets functional and vulnerability constraints. The generated results are then verified through a multi-layered closed loop consisting of Slither, Mythril, and Hardhat to ensure the authenticity, exploitability, and concealment of the vulnerabilities. The entire experimental process is managed uniformly by MLflow, enabling reproducible tracking of parameters, metrics, and generated samples.

3.2 Results Analysis

3.2.1 Component Analysis

On a knowledge base containing 10,448 structured vulnerability knowledge units, pure dense retrieval (CodeBERT), pure sparse retrieval (BM25), and hybrid retrieval (weighted fusion, $\alpha = 0.7$) are compared.

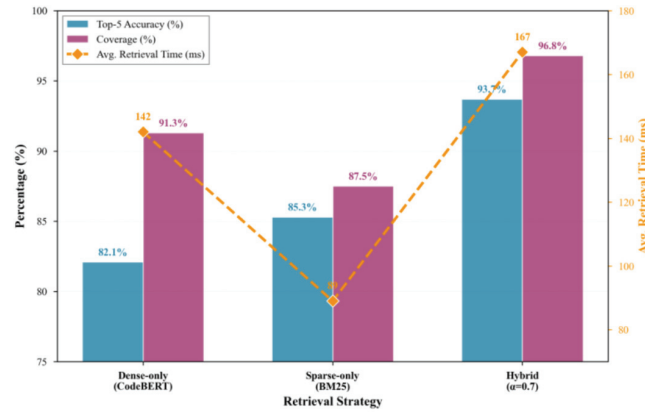


Figure 4 Component analysis.

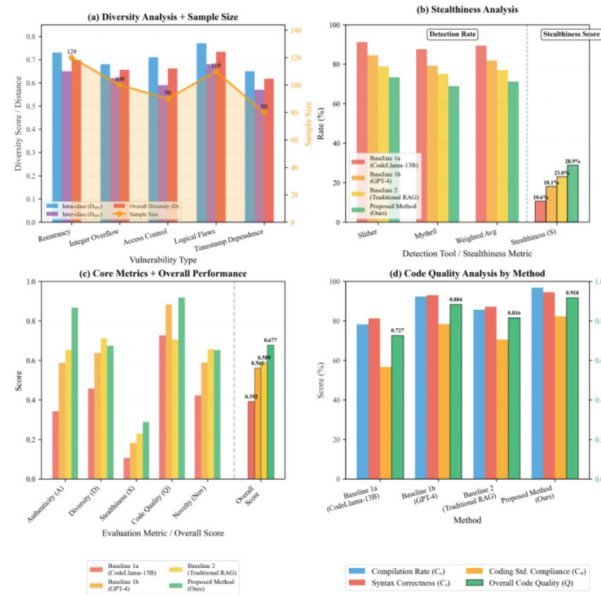


Figure 5 Generation quality analysis. (a) Diversity analysis. 5(b) Concealment analysis. (c) Comprehensive index analysis. (d) Weight distribution of each evaluation index.

It can be seen from Figure 4 that the hybrid retriever (Hybrid, $\alpha = 0.7$) excels in key quality metrics, obtaining Top-5 accuracy of 93.7%, outperforming the dense retriever (82.1%) and the sparse retriever (85.3%). Meanwhile, the hybrid retriever's coverage reaches an impressive 96.8%, remarkably better than the dense retriever (91.3%) and the sparse retriever (87.5%), meaning that it can better guarantee the availability of relevant knowledge support to fulfill various vulnerability generation tasks. Even though the average latency of the hybrid retriever is 167 ms, surpassing that of the sparse retriever (89 ms), it has compensated for this latency cost in terms of accuracy and coverage gain, making these complementary advantages of this hybrid retriever approach in semantic understanding and keyword matching.

3.2.2 Overall Performance Comparison with Baseline

In order to examine the performance of the proposed approach based on the richness of vulnerability expression and evasion capacity for vulnerability detection, a total of 500 samples of vulnerabilities were produced using the proposed approach, covering five common SWC vulnerabilities

(reentrancy attack, integer overflow, access control vulnerability, logic errors, and timestamp dependency), as shown in Figure 5.

Figure 5(a) shows that different vulnerability types exhibit significant differences in feature richness. Logic vulnerabilities exhibited the highest overall diversity (Overall_D=0.734) and outstanding intra-class diversity (D_intra=0.770), indicating their potential for expression in business rule design and logical process orchestration, providing ample space for generating diverse vulnerability variants. In contrast, timestamp-dependent vulnerabilities showed the lowest overall diversity (Overall_D=0.618), with both intra-class (0.650) and inter-class (0.570) diversity remaining at relatively low levels. This is due to the technical nature of these vulnerabilities, which rely on fixed blockchain characteristics such as block timestamps and block hashes, resulting in relatively simple vulnerability patterns. While reentrancy attacks constitute the largest category in terms of sample size (120), their overall diversity (0.698) is at a moderate level, although their inter-class diversity is only 0.650, lower than that of logic vulnerabilities (0.680).

Table 3 Comparison of ablation experiment results.

Model Variant	Authenticity (A)	Diversity (D)	Stealthiness (S)	Code Quality (Q)	Novelty (Nov)	Overall Score
Full Model (Ours)	0.867	0.674	0.289	0.918	0.652	0.677
w/o Structured KB	0.712	0.633	0.241	0.845	0.523	0.584
w/o Hybrid Retrieval	0.798	0.658	0.268	0.873	0.587	0.633
w/o Constrained Prompt	0.743	0.602	0.185	0.824	0.498	0.566
w/o Verification Loop	0.654	0.578	0.167	0.791	0.432	0.518

Figure 5(b) shows the comparison results for detection rates; from this graph, it is clearly evident that in both mainstream vulnerability detection platform tests, that the detection rate obtained by the proposed approach is significantly lower than that of all baselines: for Slither, it is reduced from 91.2% for Baseline 1a to 73.4%, and for Mythril, it decreases from 87.6% to 68.9%. Weighted average detection rate comparison clearly proves that the proposed approach with 71.2% detection is significantly lower than that of Baseline 1a (89.4%), Baseline 1b (81.9%), and Baseline 2 (77.0%). The weighted average concealment is 28.9% for the proposed approach, outperforming Baseline 1a, Baseline 1b, and Baseline 2.

Figure 5(c) is a structure showing the comparison of core metrics to indicate the effectiveness of the proposed system in achieving the five core attributes: authenticity (A), diversity (D), concealment (S), code quality (Q), and novelty (Nov). The system achieves high scores for all five attributes but performs best in terms of the measure of authenticity with a score of 0.867, registering a significant 153.5% relative improvement over Baseline 1a with a score of 0.342 and a relative improvement of 32.6% in comparison to Baseline 2 with a score of 0.654. The measure of the overall code quality with a score of Q=0.918 also registers a high degree of effectiveness in being capable of maintaining its effectiveness as a vulnerability attack, and possessing a highly superior engineering usability capability as well. The ability metric Nov score for this vulnerability is 0.652, slightly lower than Baseline 2's 0.657, but relatively better than Baseline 1a's 0.423 and Baseline 1b's 0.589. This is reflected in the high average editing distance between its code and predefined patterns in the knowledge base, as well as the high similarity of its code. The measure of the overall effectiveness with a score of Overall=0.677 validates the score in a holistic manner to register its comprehensive supremacy.

Figure 5(d) shows that of the three key engineering metrics, the proposed method outperforms the others in terms of compilation pass rate (96.8%), syntactical accuracy (94.5%), and coding standard compliance (82.3%), indicating that the generated vulnerabilities can be directly used in practical scenarios such as software testing. The normalized score for overall code quality (Q=0.918) is higher than those of all baseline methods, demonstrating that the proposed method achieves a good balance between vulnerability effectiveness and code engineering quality.

3.2.3 Ablation Experiment Analysis

To quantitatively evaluate the contribution of each key component in the proposed method, ablation experiments were conducted, and designed to progressively remove or

replace core components in the RAG framework. The performance of each ablation variant was evaluated on 200 vulnerability patterns in the Mitigation Dataset. Five ablation variants were designed so as to progressively remove key components from the full model:

Full Model: The complete method presented in this paper;

Unstructured Knowledge Base (w/o Structured KB) uses the raw vulnerability contract text instead of structured knowledge units;

The ablation variants were performed using only CodeBERT intensive retrieval (without hybrid retrieval), removing BM25 sparse retrieval; they use a simple prompt template (without constrained prompts), removing dynamic constrained prompts; and they perform only compilation checks after generation (without verification loop), removing detection and feedback iterations. Table 3 shows the performance of each ablation variant on the core evaluation metrics.

According to Table 3, the removal of the structured knowledge base resulted in a decrease in authenticity from 0.867 to 0.712, and a decrease in the overall score from 0.677 to 0.584. These results show that a well-structured representation of knowledge is essential for the generation of high-quality vulnerability codes. After the removal of the mixed retrieval strategy, the values of diversity and novelty decreased by 0.016 (from 0.674 to 0.658) and 0.065 (from 0.652 to 0.587), indicating that the strategy of sparse retrieval is inestimable in the identification of the keyword characteristics.

After discarding the constraint prompts, the concealment index decreased from 0.289 to 0.185, demonstrating that dynamic prompt templates play an essential role in assisting the model to produce vulnerable code with evasion ability. After discarding the verification loop, the authenticity index decreased from 0.867 to 0.654, indicating that the code compilation detection feedback for generating compilation is essential for ensuring its exploitability.

The ablation experimental system validated the necessity and effectiveness of each key component in the proposed method. A structured knowledge base is the key to enhancing the authenticity of generation, a hybrid retrieval strategy ensures the accuracy and diversity of knowledge, a constraint prompt engineering significantly enhances the concealment of vulnerabilities, and a programmatic verification loop is the key guarantee to ensure the exploitability of vulnerabilities.

3.2.4 Iterative Optimization Process

The iterative optimization process systematically improves the quality of vulnerability generation through a closed-loop mechanism of generation, validation, and feedback. Figure 6

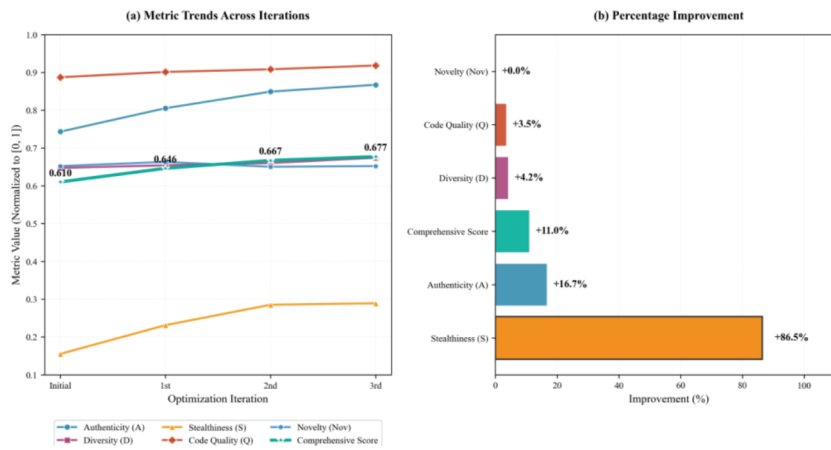


Figure 6 Iterative optimization process. (a) Each evaluation indicator undergoes 1 to 3 rounds of iterative optimization. (b) Iterative optimization increase of each evaluation indicator.

Table 4 Performance comparison before and after enhancement.

Evaluation Metric	Original Model Score	Enhanced Model Score	Statistical Significance Test
Authenticity (A)	0.763 ± 0.021	0.837 ± 0.018	$t(8) = 4.32, p < 0.001 **$
Diversity Identification Ability (D)	0.712 ± 0.024	0.798 ± 0.019	$t(8) = 3.89, p < 0.005 **$
Stealthiness Detection (S)	0.287 ± 0.019	0.313 ± 0.016	$t(8) = 3.15, p < 0.01*$
Code Quality Assessment (Q)	0.824 ± 0.017	0.856 ± 0.014	$t(8) = 2.42, p < 0.05*$
Novelty Identification Ability	0.183 ± 0.012	0.346 ± 0.022	$t(8) = 6.87, p < 0.001 ***$
Comprehensive Performance Score	0.737 ± 0.018	0.817 ± 0.015	$t(8) = 5.23, p < 0.001 ***$

shows the continuous improvement of each core evaluation indicator after three rounds of iterative optimization.

Figure 6 (a) shows that the authenticity index (A) steadily increased from an initial value of 0.743 to a final value of 0.867. This is mostly because of the continually enhanced correction of logical inconsistencies and safety mode deviations in the initial generation through the feedback mechanism, which indicates the ever-increasing comprehension of the fundamental mechanism of leakage. The stealthiness index (S) presents a large range of optimizations, increasing dramatically from 0.155 to 0.289, proving the excellent efficacy of the feedback mechanism in the augmentation of vulnerability avoidance detection.

In Figure 6 (b), stealthiness is first with a growth rate of 86.5%, directly addressing the key requirement of high-quality vulnerability generation concerning evading detection capability. During the optimization process, the diversity metric (D) showed a strong gain of 4.2%, indicating that the feedback mechanism ensured that the generated creativity was not restricted, rather it helped to create a high-quality variety of vulnerabilities with a different level of distinctiveness. The quality of the code (Q) further enhances with a growth rate of 3.5%, showing that the process is dedicated to optimizing the quality (standardization and maintainability) of the code. The novelty index (Nov) remained stable throughout the optimization process, fluctuating around 0.65, generating a baseline level of sample novelty.

3.2.5 Generate Sample Application Value Verification

To verify the practical application value of the vulnerabilities generated in this study, 1000 high-quality vulnerability

samples were added to the training set, and the security detection tool Slither was retrained in order to evaluate its performance on an independent testing set (500 real vulnerabilities). The experiment used five-fold cross validation. The comparison of performance before and after enhancement is shown in Table 4.

Table 4 shows that the vulnerability data generated by the method proposed in this paper significantly improved the performance of the detection tool in all evaluation dimensions, and all improvements reached a statistically significant level. The authenticity index increased from 0.763 to 0.837 ($t(8) = 4.32, p < 0.001$), and the 9.7% improvement verified the effectiveness of generated data in enriching vulnerability pattern learning. The novelty recognition ability increased from 0.183 to 0.346 ($t(8) = 6.87, p < 0.001$), with a significant improvement of 89.1%, directly demonstrating the unique value of generated data in expanding the detection range.

In terms of diversity and concealment identification, the improved model showed statistically significant improvements in performance of 12.1% and 9.1%, respectively ($t(8) = 3.89, p < 0.005$; $t(8) = 3.15, p < 0.01$). The improved diversity identification capability captures the significance of generated data in enhancing detection coverage; this is because this tool has improved capability in identifying various forms of implementation of the vulnerability type through learning various forms of structural variations generated in this study. The improved concealment capability indicates this tool's ability to increase sensitivity in vulnerability detection applied in well-crafted samples by learning highly concealed structures in generated samples with concealment $S = 0.313$. The improved code

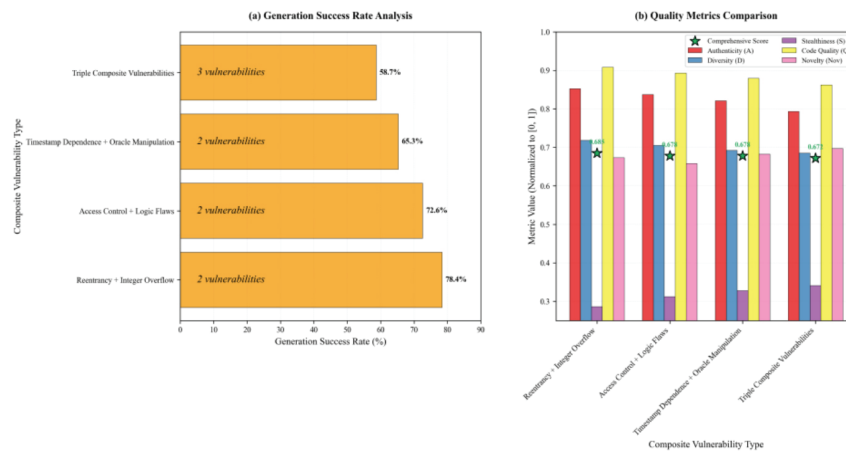


Figure 7 Evaluation metrics for composite vulnerabilities. 7(a) Success rate analysis of composite vulnerability generation. 7(b) Comparison of quality metrics for composite vulnerabilities.

quality assessment capability with 3.9%, even though less, is statistically significant ($t(8) = 2.42$, $p < 0.05$) in capturing comprehensive benefits offered by generated data in improving the assessment accuracy of this tool.

3.2.6 Case Study and Advanced Capability Demonstration

To test the ability of the proposed approach to create complex scenarios, this experiment evaluated the performance of the system when creating composite scenarios with multiple vulnerabilities. For this purpose, four types of composite vulnerabilities were used, each comprising 100 samples. The results for the evaluation are presented in Figure 7.

Figure 7(a) above shows that reentrancy + integer overflow has the highest success rate of 78.4%. This is because of their high co-occurrence rate coupled with their interaction logic for smart contracts. Notably, the integer overflow is most likely to cause unusual scenarios for fund transactions, which serve as conditions for reentrancy attacks. The success rate for access control + logic vulnerability is second highest at 72.6%, which also stems from their natural correlation for smart contracts. The time dependency + oracle manipulation has a success rate of 65.3%. The success rate for triple composite vulnerabilities is lower at 58.7%. The more complex a composite vulnerability, the lower is its success rate; however, this rate diminishes gradually to prove the applicability of the RAG framework to form reasonable composite logic by integrating multiple kinds of knowledge elements.

In Figure 7(b), the authenticity index A retains a high level in all scenarios, with a highest level of 0.852 in the reentrancy + integer overflow scenario, thereby validating the effectiveness of the vulnerability generation process as a whole. The concealment index S clearly has a rising trend and increases from .286 in the reentrancy + overflow to .341 in the triple scenario because of the collective avoidance provided by stacking a vulnerability. The complexity logic provided by this interaction logic makes it difficult to identify by a detection tool.

Structural diversity (D) slightly reduced from 0.718 (reentrancy and overflow) to 0.685 (triple composite) and still

kept a high level. Code quality (Q) slightly reduced with increasing complexity from 0.908 to 0.862 and kept over 0.86, which suggested that the system could still ensure a good code standardization and engineering usability during the integration of various vulnerabilities. Pattern novelty (Nov) reached 0.697 in triple composite vulnerabilities and showed that higher-order patterns have a good innovative expression ability.

4. CONCLUSIONS

To resolve the inconsistency between high generation ability but low domain knowledge of current large language models (LLM) for smart contract vulnerability generation, an automatic strategy using the RAG framework is presented. To demonstrate the effectiveness and advantages of the proposed strategy, a series of experiments were carried out, with results indicating that the proposed strategy can greatly increase the vulnerability code generation from 0.342 (CodeLlama (Code Large Language Model Meta AI)-13B) to 0.867, with the generated data improving the current vulnerability detectors. Furthermore, the current work promotes the paradigm shift for security studies from “passive detection” to “active generation” and offers an important strategy reference for the safe use of LLMs in intensive domains.

REFERENCES

1. Piantadosi, V., Rosa, G., Placella, D., Scalabrino, S., & Oliveto, R. (2023). Detecting functional and security-related issues in smart contracts: A systematic literature review. *Software: Practice and Experience*, 53(2), 465–495. <https://doi.org/10.1002/spe.3156>
2. Taherdoost, H. (2023). Smart Contracts in Blockchain Technology: A Critical Review. *Information*, 14(2), 117. <https://doi.org/10.3390/info14020117>
3. Khan, S. N., Loukil, F., Ghedira-Guegan, C., Benkhelifa, E., & Bani-Hani, A. (2021). Blockchain smart contracts: Applications, challenges, and future trends. *Peer-to-peer Networking and Applications*, 14(5), 2901–2925. <https://doi.org/10.1007/s12083-021-01127-0>

4. Tolmach, P., Li, Y., Lin, S. W., Liu, Y., & Li, Z. (2021). A survey of smart contract formal specification and verification. *ACM Computing Surveys (CSUR)*, 547, 1–38. <https://doi.org/10.1145/3464421>
5. He, D., Wu, R., Li, X., Chan, S., & Guizani, M. (2023). Detection of vulnerabilities of blockchain smart contracts. *IEEE Internet of Things Journal*, 1014, 12178–12185. <https://doi.org/10.1109/JIOT.2023.3241544>
6. Ivanov, N., Li, C., Yan, Q., Sun, Z., Cao, Z., & Luo, X. (2023). Security threat mitigation for smart contracts: A comprehensive survey. *ACM Computing Surveys*, 55(14s), 1–37. <https://doi.org/10.1145/3593293>
7. Weyssow, M., Zhou, X., Kim, K., Lo, D., & Sahraoui, H. (2025). Exploring parameter-efficient fine-tuning techniques for code generation with large language models. *ACM Transactions on Software Engineering and Methodology*, 347, 1–25. <https://doi.org/10.1145/3714461>
8. Taghavi Far, S. M., & Feyzi, F. (2025). Large language models for software vulnerability detection: a guide for researchers on models, methods, techniques, datasets, and metrics. *International Journal of Information Security*, 242, 78. <https://doi.org/10.1007/s10207-025-00992-7>
9. Das, B. C., Amini, M. H., & Wu, Y. (2025). Security and privacy challenges of large language models: A survey. *ACM Computing Surveys*, 576, 1–39. <https://doi.org/10.1145/3712001>
10. Guo, W., Xue, J., Liu, Z., Han, W., & Hu, J. (2025). Malgta: large language model-based guided malware tactical analysis: W. Guo et al. *The Journal of Supercomputing*, 819, 1049. <https://doi.org/10.1007/s11227-025-07545-8>
11. Boi, B., Esposito, C., & Lee, S. (2024). Smart contract vulnerability detection: The role of large language model (llm). *ACM SIGAPP Applied Computing Review*, 242, 19–29. <https://doi.org/10.1145/3687251.3687253>
12. Zaazaa, O., & El Bakkali, H. (2024). SmartLLMSentry: A Comprehensive LLM Based Smart Contract Vulnerability Detection Framework. *Journal of Metaverse*, 42, 126–137. <https://doi.org/10.57019/jmv.1489060>
13. Mandana, E., Vlahavas, G., & Vakali, A. (2025). EVuLLM: Ethereum Smart Contract Vulnerability Detection Using Large Language Models. *Electronics*, 1416, 3226. <https://doi.org/10.3390/electronics14163226>
14. Ait Hsain, Y., Laaz, N., & Mbarki, S. (2025). SCEditor-Web: Bridging Model-Driven Engineering and Generative AI for Smart Contract Development. *Information*, 1610, 870. <https://doi.org/10.3390/info16100870>
15. Kiani, R., & Sheng, V. S. (2024). Ethereum Smart Contract Vulnerability Detection and Machine Learning-Driven Solutions: A Systematic Literature Review. *Electronics*, 1312, 2295. <https://doi.org/10.3390/electronics13122295>
16. Liu, Z., Qian, P., Wang, X., Zhuang, Y., Qiu, L., & Wang, X. (2021). Combining graph neural networks with expert knowledge for smart contract vulnerability detection. *IEEE Transactions on Knowledge and Data Engineering*, 352, 1296–1310. <https://doi.org/10.1109/TKDE.2021.3095196>
17. Wu, H., Dong, H., He, Y., & Duan, Q. (2023). Smart Contract Vulnerability Detection Based on Hybrid Attention Mechanism Model. *Applied Sciences*, 132, 770. <https://doi.org/10.3390/app13020770>
18. Kannengiesser, N., Lins, S., Sander, C., Winter, K., Frey, H., & Sunyaev, A. (2021). Challenges and common solutions in smart contract development. *IEEE Transactions on Software Engineering*, 4811, 4291–4318. <https://doi.org/10.1109/TSE.2021.3116808>
19. Li, X., Wang, H., Liu, Z., Yu, S., Wang, S., Yan, Y., ... & Yu, G. (2025). Building a coding assistant via the retrieval-augmented language model. *ACM Transactions on Information Systems*, 432, 1–25. <https://doi.org/10.1145/3695868>
20. Xiang, J., Gao, Z., Bao, L., Hu, X., Chen, J., & Xia, X. (2025). Automating comment generation for smart contract from bytecode. *ACM Transactions on Software Engineering and Methodology*, 343, 1–31. <https://doi.org/10.1145/3699597>
21. Yong, C., Defeng, H., Chao, X., Nannan, C., & Jianbo, L. (2025). Smart contract generation model based on code annotation and AST-LSTM tuning. *The Journal of Supercomputing*, 815, 731. <https://doi.org/10.1007/s11227-025-07186-x>
22. Misbah, S., Shahid, M. F., Siddiqui, S., Khanzada, T. J. S., Ashari, R. B., Ullah, Z., & Jamjoom, M. (2025). Generative AI-Driven Smart Contract Optimization for Secure and Scalable Smart City Services. *Smart Cities*, 84, 118. <https://doi.org/10.3390/smartcities8040118>
23. Kado, C., Yanai, N., Cruz, J. P., Yamashita, K., & Okamura, S. (2025). Empirical Study of Impact of Solidity Compiler Updates on Vulnerabilities in Ethereum Smart Contracts. *Distributed Ledger Technologies: Research and Practice*, 42, 1–14. <https://doi.org/10.1145/3688812>
24. Tang, X., Du, Y., Lai, A., Zhang, Z., & Shi, L. (2023). Deep learning-based solution for smart contract vulnerabilities detection. *Scientific Reports*, 131, 20106. <https://doi.org/10.1038/s41598-023-47219-0>
25. Fang, M., Hu, H., Hu, F., & Liu, J. (2025). A Survey of the Full Process of Code Search Based on Deep Learning. *Concurrency and Computation: Practice and Experience*, 37(23–24), e70277. <https://doi.org/10.1002/cpe.70277>
26. Lamsiyah, S., El Mahdaoui, A., Ouatik El Alaoui, S., & Espinasse, B. (2023). Unsupervised query-focused multi-document summarization based on transfer learning from sentence embedding models, BM25 model, and maximal marginal relevance criterion. *Journal of Ambient Intelligence and Humanized Computing*, 143, 1401–1418. <https://doi.org/10.1007/s12652-021-03165-1>
27. Sfyrakis, I., Modesti, P., Golightly, L., & Ikegima, M. (2025). LightCross: A Lightweight Smart Contract Vulnerability Detection Tool. *Computers*, 149, 369. <https://doi.org/10.3390/computers14090369>
28. Jiao, T., Xu, Z., Qi, M., Wen, S., Xiang, Y., & Nan, G. (2024). A survey of Ethereum smart contract security: Attacks and detection. *Distributed Ledger Technologies: Research and Practice*, 33, 1–28. <https://doi.org/10.1145/3643895>
29. Yu, Y., Rong, G., Shen, H., Zhang, H., Shao, D., Wang, M., ... & Wang, J. (2024). Fine-tuning large language models to improve accuracy and comprehensibility of automated code review. *ACM Transactions on Software Engineering and Methodology*, 341, 1–26. <https://doi.org/10.1145/3695993>
30. Ren, X., & Wei, Q. (2024). Sligpt: A Large Language Model-Based Approach for Data Dependency Analysis on Solidity Smart Contracts. *Software*, 33, 345–367. <https://doi.org/10.3390/software3030018>
31. Odofoin, T., Abayomi, A. A., Ogbuefi, E., Ogeawuchi, J. C., Adanigbo, O. S., & Gbenle, T. P. (2024). Strategic integration of LangChain, Hugging Face Transformers, and OpenAI for document intelligence systems. *International Journal of Scientific Research in Science Engineering and Technology*, 114, 413–426. <https://doi.org/10.32628/IJSRSET25121177>
32. Sipola, T., Juvonen, A., & Lehtonen, J. (2012). Dimensionality reduction framework for detecting anomalies from network logs. *Engineering Intelligent Systems*, 1(2).
33. Zipperle, M., Becherer, M., Zhang, Y., Chang, E., Dillon, T., & Karduck, A. (2023). Enabling Digital Transformation with Sustainability Criteria through Resilient Cybersecurity: Challenges and Opportunities. *Engineering Intelligent Systems*, 316, 1–7.