Wang *et al.* *J Surveill Secur Saf* 2024;5:258-72
DOI: 10.20517/jsss.2024.21

Journal of Surveillance,
Security and Safety

Research Article

Open Access

VulnScan GPT: a new framework for smart contract vulnerability detection combining vector database and GPT model

Lianjie Wang, Yunhao Zhao, Keqing Wang, Zhongwen Zhang, Wenyin Zhang

School of Computer Science & Engineering, Linyi University, Linyi City, Shandong Province, 276000, China.

Correspondence to: Wenyin Zhang, Zhongwen Zhang, School of Computer Science & Engineering, Linyi University, Industrial Avenue (North Section, West Side), Lanshan District, Linyi City, Shandong Province, 276000, China. E-mail: zhangwenyin@lyu.edu.cn; zhangzhongwen@lyu.edu.cn

How to cite this article: Wang L, Zhao Y, Wang K, Zhang Z, Zhang W. VulnScan GPT: a new framework for smart contract vulnerability detection combining vector database and GPT model. *J Surveill Secur Saf* 2024;5:258-72. http://dx.doi.org/10.20517/jsss.2024.21

Received: 30 Aug 2024 **First Decision:** 19 Nov 2024 **Revised:** 29 Nov 2024 **Accepted:** 10 Dec 2024 **Published:** 31 Dec 2024

Academic Editor: Panayiotis Kotzanikolaou **Copy Editor:** Ting-Ting Hu **Production Editor:** Ting-Ting Hu

Abstract

With the rapid development of blockchain technology and smart contracts, the security issues of smart contracts have become increasingly serious. To address the significant limitations of traditional detection methods in handling the complexity and scale of smart contracts, a new framework for smart contract vulnerability detection that combines a vector database and a generative pre-trained transformer (GPT) model — VulnScan GPT — has been proposed. This framework comprises three main components: function signature extraction, vector database storage and retrieval, and GPT-based vulnerability detection. The framework uses the solc tool to generate an abstract syntax tree from smart contracts, extract function signatures, and vectorize the code for storage. By integrating the GPT model, the framework can preliminarily analyze and filter key functions based on common vulnerability scenarios and then retrieve relevant implementations from the vector database for in-depth assessment. This method gradually optimizes function analysis through an iterative detection mechanism, leveraging the efficient storage and retrieval capabilities of the vector database, combined with the deep natural language processing abilities of the GPT model, enhancing the accuracy and comprehensiveness of vulnerability detection. In the experimental evaluation, tests were conducted on various datasets to detect automated market maker price manipulation and initial risk deposit vulnerabilities, demonstrating that VulnScan GPT not only improves the accuracy of vulnerability detection, but also significantly reduces operational costs by optimizing token usage, resulting in an efficient and cost-effective detection solution.

Keywords: ChatGPT, iterative detection, VulnScan GPT, smart contract, vector database, token efficiency

1. INTRODUCTION

Since the advent of Bitcoin and its underlying blockchain technology in 2008, blockchain applications and cryptocurrencies have undergone significant development. Especially for smart contract platforms such as Ethereum, whose market value exceeded $400 billion in 2023. With the widespread application of smart contracts, their security issues have significantly increased. Particularly after The Decentralized Autonomous Organization (DAO) suffered a reentrancy attack in 2016, the industry has paid more attention to security vulnerabilities caused by human errors and design flaws in smart contracts. Unlike traditional software functional errors, vulnerabilities in smart contracts usually lead to direct financial losses, making their detection especially complex and requiring in-depth analysis of specific domain attributes.

A recent study[1] conducted a systematic investigation of 167 smart contracts with real vulnerabilities on the Code4rena platform from 2021 to 2022, finding that over 80% of the vulnerabilities were so-called machine-unverifiable bugs (MUBs) that existing tools could not detect. Due to the traditional analysis methods' lack of understanding of smart contract semantics, they find it difficult to grasp the complex relationships between code vulnerabilities and associated attack behaviors. Zhang *et al.*.[1] categorize smart contract vulnerabilities by their detection needs into two groups: those detectable with simple and general test oracles, and those requiring advanced semantic oracles for detection. Vulnerabilities requiring advanced semantic oracles include price oracle manipulation, ID-related violations, state update errors, atomicity violations in business processes, privilege escalation and access control issues, accounting errors, and business model disruptions. Detecting these vulnerabilities requires higher-level semantic analysis to identify the complex relationships and potential issues between code and business logic.

Generative Pre-trained Transformer (GPT)[2,3] is an advanced natural language processing model developed by OpenAI. Through large-scale pre-training and fine-tuning, it generates high-quality text, understands and produces human language, and performs complex language understanding and generation tasks. Due to its powerful semantic analysis capabilities, GPT can effectively assist in contract vulnerability detection[4]. However, applying GPT directly to extensive smart contract code can increase token consumption and potential overload, reducing the model's efficiency and accuracy. This limitation is crucial as it reflects on the scalability challenges of current methods when managing large-scale contract audits.

Therefore, when dealing with large amounts of code and complex data structures, a system capable of efficiently managing and retrieving information is required. Vector databases[5], a technology specifically designed for storing and retrieving high-dimensional vector data, perfectly meet this need. Converting the features of smart contracts into vector form enables rapid and accurate similarity searches and data comparisons, thereby greatly supporting various vulnerability detection applications. Additionally, the core functions of vector databases, including efficient similarity searches, vector indexing, and data storage and management, provide the necessary support for the GPT model when processing smart contract code.

This study introduces VulnScan GPT, a novel smart contract detection framework that combines the vector database and the natural language processing capabilities of the GPT model to overcome the limitations of existing methods and improve the accuracy and comprehensiveness of vulnerability detection. VulnScan GPT utilizes the efficient storage and retrieval capabilities of the vector database to comprehensively manage smart contract code without being constrained by token length limitations. The system performs vectorization to quickly retrieve the contextual information needed by the GPT model, preserving the semantic integrity of the code and avoiding detection issues caused by context loss or fragmented processing. Through this innovative framework, we aim to significantly enhance the security detection of smart contracts, thereby protecting the safety and stability of blockchain applications. The structure of this paper is organized as follows: Section 2 reviews related studies and analyzes the limitations of existing approaches. Section 3 provides a detailed introduction to the framework design of VulnScan GPT. Section 4 evaluates its performance. Finally, Section 6 discusses potential improvements and summarizes the research contributions.

2. RELATED WORKS

Traditional smart contract vulnerability detection methods mainly include static analysis, dynamic analysis, symbolic execution, and formal verification. Static analysis tools, such as Vandal[6], Securify[7], and Slither[8], detect potential vulnerabilities by analyzing the structure and logic of smart contract code. Dynamic analysis tools, such as Echidna[9], ContractFuzzer[10], and Harvey[11], use fuzz testing techniques to generate test inputs and identify anomalies during the actual execution of smart contracts, thereby gaining insights into runtime vulnerabilities. Symbolic execution tools, such as Oyente[12], Halmos[13], and Mythril[14], detect vulnerabilities by simulating contract execution paths and checking possible states along each path. Formal verification tools, such as ZEUS[15], VerX[16], and VeriSmart[17], use rigorous mathematical proofs to verify the correctness and security of smart contracts. However, research by Zhang *et al.*[1] found that traditional analysis methods lack an understanding of smart contract semantics, making it difficult to grasp the complex relationships between code vulnerabilities and related attack behaviors.

The rapid advancement of deep learning has opened up new opportunities[18,19]. With the rapid development of language models, the application of the GPT series models in code-related tasks has also increased. GPT models, particularly GPT-3 and GPT-4, have been extensively studied and utilized for code repair and vulnerability detection tasks[20].

David *et al.*[21] evaluated the performance of GPT-4 in smart contract auditing, showing that it has a certain level of accuracy in identifying smart contract vulnerabilities. In some cases, the GPT-4 model correctly identified the types of vulnerabilities, achieving a true positive rate of 78.7%. Existing research typically focuses on directly using GPT for code vulnerability detection. Although this approach is feasible for smaller amounts of code, it can lead to a significant increase in token usage when dealing with large contract projects. Moreover, inputting all code information into GPT is not optimal as it increases meaningless token consumption and can cause information overload, leading to confusion in the large language model and impacting its performance.

GPTLens[22] proposed an innovative two-stage framework by designing large language models (LLMs) to play the antagonistic roles of auditor and commentator, significantly improving the performance of traditional single-stage detection methods. This approach addressed the accuracy challenges LLMs face when dealing with real-world datasets due to high false positive rates. However, GPTLens also faces similar issues as those identified in research by David *et al.*[21], struggling to effectively handle contract projects with large amounts of code.

GPTScan combines the GPT model with static analysis, breaking down logical vulnerabilities into Scenarios and Properties for matching, thereby improving the accuracy and efficiency of smart contract vulnerability detection. This method employs multidimensional filtering strategies (including project-wide files, OpenZeppelin function filtering, etc.) to accurately screen candidate functions, effectively reducing false positives. However, its heavy reliance on predefined rules and static analysis may limit its adaptability to new or complex vulnerabilities, while excessive pre-filtering might overlook necessary contextual information, affecting the comprehensiveness of detection. Additionally, the reachability analysis process may cause unnecessary duplicate detections, increasing resource consumption.

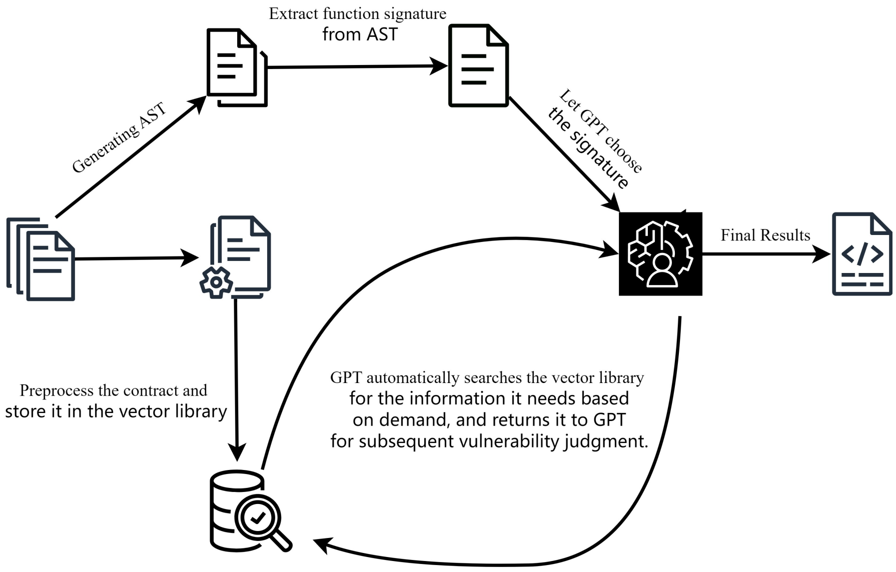

Extract function signature from AST
Generating AST
Let GPT choose the signature
Final Results
Preprocess the contract and store it in the vector library
GPT automatically searches the vector library for the information it needs based on demand, and returns it to GPT for subsequent vulnerability judgment.

Figure 1. VulnScan GPT process overview.

Our proposed VulnScan GPT framework enhances detection specificity and efficiency by extracting function signatures to precisely locate code related to specific vulnerabilities. Through dynamic interaction with the vector database, the system automatically searches for and further inspects the code after initial analysis, ensuring comprehensive and accurate detection through iterative detection processes. VulnScan GPT is more effective than other methods in handling large-scale codebases, adapting to complex and dynamic coding environments, and significantly improving detection efficiency and accuracy.

3. VULNSCAN GPT

This study aims to address the growing demand for vulnerability detection in smart contracts prior to deployment by designing a framework that integrates the capabilities of the GPT model with vector databases. Specifically, our approach seeks to enhance the efficiency of static analysis processes, minimizing both detection time and cost, and thus contributing to the acceleration of the security auditing workflow.

In this chapter, we provide a detailed introduction to the design and implementation of the VulnScan GPT framework. The goal of this system is to automatically detect potential vulnerabilities in smart contracts by combining Abstract Syntax Tree (AST) analysis and vectorized storage technology with the powerful understanding capabilities of GPT. The process of VulnScan GPT is shown in Figure 1.

Next, we will introduce the three main components of VulnScan GPT: function signature extraction, vector database storage and retrieval, and GPT-based vulnerability detection.

Function Signature Extraction: Using the solc tool to generate ASTs for smart contracts[23], extracting the signature of each function. However, we do not extract signatures from all files. For example, interface files (i.e., files where the ”contracting” key in the AST tree corresponds to ”interface”), files in the ”node_modules” directory, test files (e.g., those found in various ”test” directories), and third-party library files (e.g., those from well-known libraries such as ”OpenZeppelin,” ”Uniswap,” and ”PancakeSwap”). Once these files are filtered out, we can focus on the project’s files.

Vector Database Storage and Retrieval: Considering that when using GPT to generate supplementary information, the generated function signature has a certain degree of ambiguity (such as parameter types are similar but not exactly the same), and may not be completely consistent with the actual function, we choose to use the cosine similarity of the vector database to find the most similar function. The contracts of the target project are vectorized and stored in the vector database. This process not only includes the vectorization of contracts but also involves removing interface declaration files and test files to ensure that subsequent vector retrieval is not disrupted, thereby improving the accuracy and relevance of retrieval results.

GPT-Based Vulnerability Detection: The extracted function signatures are analyzed by GPT, which automatically selects the functions to be inspected based on the vulnerabilities to be detected and their common scenarios, making an initial judgment. Subsequently, GPT-4 performs an in-depth analysis of these functions. Through iterative multi-round detection and scoring mechanisms, it dynamically adjusts the query strategy, gradually refining the assessment until sufficient information supports a definitive vulnerability judgment.

3.1. AST generation and function signature extraction

First, we use solc-js to compile the project's contracts to obtain the AST for each contract. From the generated AST, we extract each function's declaration and signature and store this information. At the same time, we also obtain the ASTs corresponding to the external libraries referenced in the project. However, the ASTs generated from these external libraries and interface files are not needed. Therefore, we employ methods to skip these and focus on extracting functions from the project files.

In the generated AST, we determine whether to skip specific files by checking the 'absolutePath' property of the top-level node. If the path contains "openzeppelin" or is not within the path where we store the contract project, we directly skip these AST files and do not extract function signatures from them.

We also choose to skip files where the 'contractKind' attribute in the AST node is 'interface'. These interface files only contain function declarations but do not include function implementations. Function declarations in interface files can all be called externally, but not all function signatures will be declared in interface files. Therefore, relying solely on interface files for signature extraction is insufficient. The content of interface files may overlap with the function signatures we extract from the AST tree, adding unnecessary redundancy. By these means, we can more efficiently focus on the key functions in the project, avoiding unnecessary interference from external libraries and interface files, thus improving the accuracy and efficiency of function signature extraction.

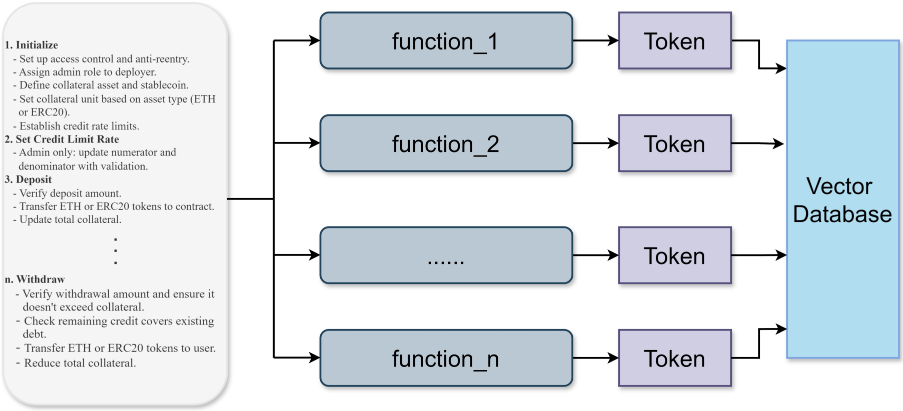
3.2. Vector database storage and retrieval

In VulnScan GPT, vector database storage and retrieval are crucial components. This section will describe in detail our design choices for the vector database storage and retrieval component. A vector database stores the code functions of smart contracts as high-dimensional vectorized chunks, allowing GPT to focus on context-relevant functional blocks and avoiding the computational resource waste associated with processing the entire contract code directly. GPT then analyzes this data to further uncover potential vulnerability patterns. The vectorization process is shown in Figure 2.

3.2.1. Vectorization process

In the vectorization process, we first perform feature extraction on the raw data and convert it into high-dimensional vectors. This process is crucial because high-quality vectorization can significantly improve the accuracy and efficiency of retrieval.

Letting the data to be vectorized be$f$, its vector representation is V_f, which can be given as



1. Initialize
- Set up access control and anti-reentry.
- Assign admin role to deployer.
- Define collateral asset and stablecoin.
- Set collateral unit based on asset type (ETH or ERC20).
- Establish credit rate limits.

2. Set Credit Limit Rate
- Admin only: update numerator and denominator with validation.

3. Deposit
- Verify deposit amount.
- Transfer ETH or ERC20 tokens to contract.
- Update total collateral.

⋮

n. Withdraw
- Verify withdrawal amount and ensure it doesn't exceed collateral.
- Check remaining credit covers existing debt.
- Transfer ETH or ERC20 tokens to user.
- Reduce total collateral.

Figure 2. We take a sol file in the project as an example, split it into n function-level code blocks, and vectorize them separately and store them in the vector database.

$$V_f = \text{embedding}(f) \tag{1}$$

Where embedding() represents the embedding model, mapping the function to a high-dimensional vector.

3.2.2. Segmentation strategy

We employ a method of vectorizing the smart contract code, converting each function into a fixed-length high-dimensional vector. At the same time, we simultaneously choose to segment the smart contract code at the function level. If the block is too small, it will cause the complete function to be split into different segments, making the semantic information of the function code incomplete. If the block is too large, excessive overlapping information may lead to meaningless token wastage. Segmentation at the function level helps enhance the granularity of vector representation, allowing each vector to precisely represent the semantic information of a single function. Additionally, we will add the file address and ”start_index” as ”metadata” during storage, so that when we subsequently use function signatures for vector similarity calculations and searches, we can obtain the required function bodies more quickly and accurately.

3.2.3. Storage strategy

To store and manage the generated high-dimensional vectors, we chose a local vector database. The local vector database uses an efficient local storage mechanism, enabling quick access to vector data. This is crucial for the smart contract vulnerability detection system, which requires frequent vector retrievals, ensuring high response speed for the system.

To ensure the efficiency and accuracy of the vector database, we conduct preliminary screening and reduction work before storing project contracts in the vector database. Interface files, test files, and some runtime-generated non-Solidity files and non-documentation files are the files we do not need to store. This approach not only reduces storage pressure but also improves the overall efficiency of the system. Interface files need to be deleted because they only contain various function signature declarations and do not include complete function implementations. These function signatures may interfere with search results when searching for function implementations through vector similarity, affecting accuracy. Therefore, we choose not to store these interface files. Test files, runtime-generated non-Solidity files, and non-documentation files are also not within the storage scope. These files usually contain test cases and temporarily generated data, which do not directly aid in the implementation and analysis of contract functionality, but rather increase the system’s storage and processing burden.

3.2.4. Content retrieval

First, we generate query vectors based on the characteristics of the functions to be detected. Then, we perform similarity searches using the vector database, determining their similarity by calculating the cosine similarity between the query vectors and the stored vectors in the database. The cosine distance is calculated by:

$$d = 1.0 - \frac{\sum(A_i \times B_i)}{\sqrt{\sum(A_i^2) \times \sum(B_i^2)}} \tag{2}$$

Where A_i and B_i represent the components of the two vectors, and d is the cosine distance, obtained by subtracting the cosine similarity from 1.

By calculating the cosine distance, we can obtain the most similar blocks. These vectors represent code segments in the database that are most similar to the target function. The returned results include similar vectors and their corresponding metadata, such as the function body and file path.

Due to the limitations of the segmentation method or the results returned by vector retrieval, the returned content may only be a part of the code and may not be a complete function. In such cases, we need to assess the returned code blocks and complete any incomplete functions. The specific steps are as follows:

Upon receiving the retrieved code blocks, we first determine whether they constitute a complete target function. If the code block is complete, it is directly passed to GPT for analysis. If the code block is incomplete, it needs to be supplemented. We use the ”start_index” index-based method for supplementation. By locating the file position, we gradually add subsequent content in fixed sizes until the target function is complete. To avoid excessive code blocks bringing redundant information and affecting GPT’s judgment while saving tokens, we choose the smallest possible size for the supplementary blocks. This strategy not only ensures the efficiency of the supplementation process, but also improves the accuracy and efficiency of GPT’s analysis.

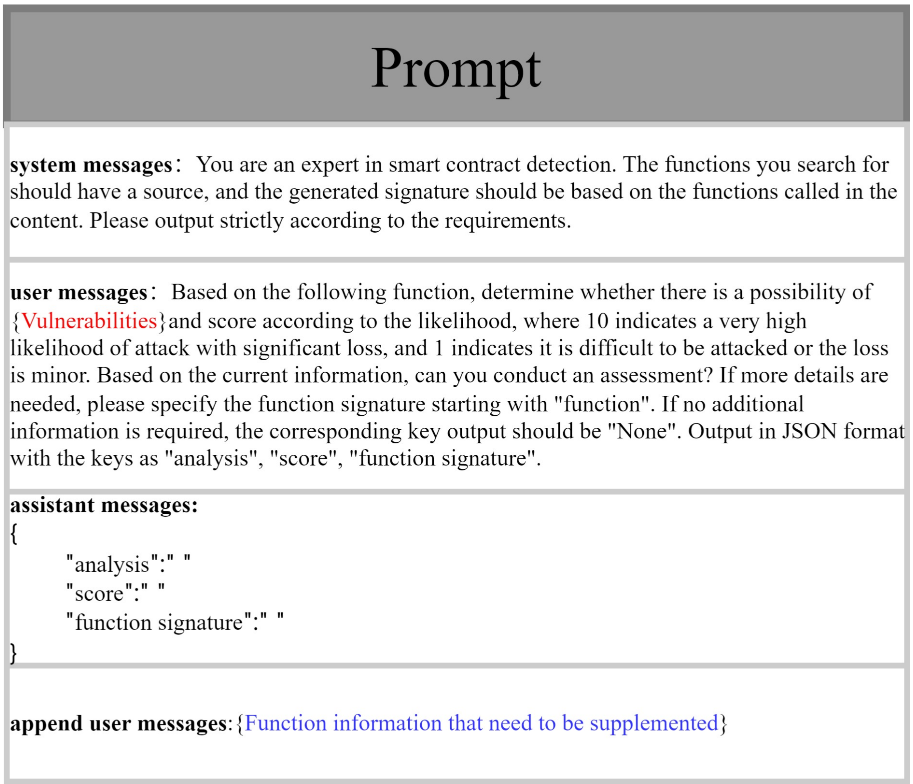
3.3. GPT-based vulnerability detection

The system consolidates all extracted function signatures into a JSON format and submits it to GPT for initial assessment. Based on the description of specific vulnerabilities and common scenarios, GPT will select the functions that need to be checked and return their signatures. The system then retrieves the corresponding function implementations from the vector database for further assessment.

First Phase: Utilize the GPT-4 model to automate the pre-screening process of smart contract vulnerabilities. In this process, the model initially selects functions based on the description of a specific vulnerability and its occurrence in common function scenarios, outputting the function signatures to be checked for subsequent matching in the vector database. This screening method effectively reduces the range of potential risk functions before we proceed to code-level vulnerability detection. It also significantly reduces the number of functions required for vector search and GPT model detection, thereby saving computational resources and OpenAI tokens, and enhancing the efficiency and cost-effectiveness of the detection process.

Second Phase: Conduct Iterative Detection and Scoring. We use prompts to impose format constraints and have GPT make judgments on the complete functions identified in the first phase through the vector database. The screened complete functions are submitted to GPT for review, where it analyzes the current content, scores it, and searches for the required information. Figure 3 shows a process of using prompts in this phase.

In this process, the GPT model may produce two outcomes: one is that the existing function information is sufficient to diagnose potential issues; the other is that the existing information is insufficient to make an accurate judgment. For the latter case, the system implements an iterative multi-round detection mechanism.



Prompt

system messages: You are an expert in smart contract detection. The functions you search for should have a source, and the generated signature should be based on the functions called in the content. Please output strictly according to the requirements.

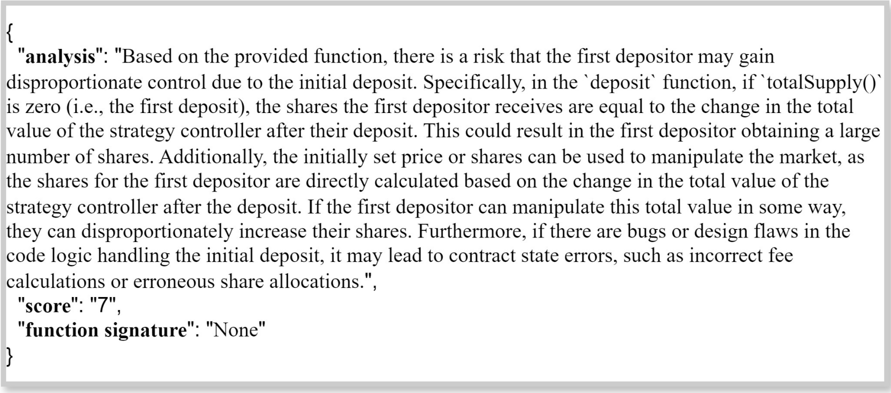
user messages: Based on the following function, determine whether there is a possibility of {Vulnerabilities}and score according to the likelihood, where 10 indicates a very high likelihood of attack with significant loss, and 1 indicates it is difficult to be attacked or the loss is minor. Based on the current information, can you conduct an assessment? If more details are needed, please specify the function signature starting with "function". If no additional information is required, the corresponding key output should be "None". Output in JSON format with the keys as "analysis", "score", "function signature".

assistant messages:

```
{
        "analysis":" "
        "score":" "
        "function signature":" "
}
```

append user messages:{Function information that need to be supplemented}

Figure 3. VulnScan GPT framework detection and scoring prompt.



```
{
  "analysis": "Based on the provided function, there is a risk that the first depositor may gain
disproportionate control due to the initial deposit. Specifically, in the `deposit` function, if `totalSupply()`
is zero (i.e., the first deposit), the shares the first depositor receives are equal to the change in the total
value of the strategy controller after their deposit. This could result in the first depositor obtaining a large
number of shares. Additionally, the initially set price or shares can be used to manipulate the market, as
the shares for the first depositor are directly calculated based on the change in the total value of the
strategy controller after the deposit. If the first depositor can manipulate this total value in some way,
they can disproportionately increase their shares. Furthermore, if there are bugs or design flaws in the
code logic handling the initial deposit, it may lead to contract state errors, such as incorrect fee
calculations or erroneous share allocations.",
  "score": "7",
  "function signature": "None"
}
```

Figure 4. An example of the test result output, equivalent to the assistant messages section in Figure 3.

We show one example of this judgment in Figure 4. When the ”function signature” is ”None”, we consider the detection complete, meaning the current information is sufficient for GPT to make a judgment.

When the GPT model indicates that more information is needed, the system queries the vector database based on the specific function signatures provided by GPT. This query aims to obtain the missing function implementations or related data, which are then resubmitted to GPT for further evaluation. The addition of new information each time can change the accuracy and results of the evaluation. Therefore, this multi-round iterative process continues until the GPT model confirms that the retrieved function information is sufficient to support a clear judgment, which is indicated by a ”None” supplementary signature, with the final scoring as the criterion. Through this adaptive selection mechanism, GPT can automatically identify and supplement the key contextual information required to handle specific tasks with minimal code. This method optimizes the efficiency of information processing and saves the cost of using related tokens for detection. Figure 2 shows the prompt used by the VulnScan GPT framework during the detection phase.

This multi-round iterative detection mechanism enables the system to dynamically adjust query strategies, gradually improving the analysis and evaluation of each function. This mechanism not only increases the accuracy of vulnerability detection but also greatly enhances the flexibility and adaptability of the process. Ultimately, through continuous information supplementation and GPT analysis, the system can effectively identify and prevent potential vulnerabilities in smart contracts.

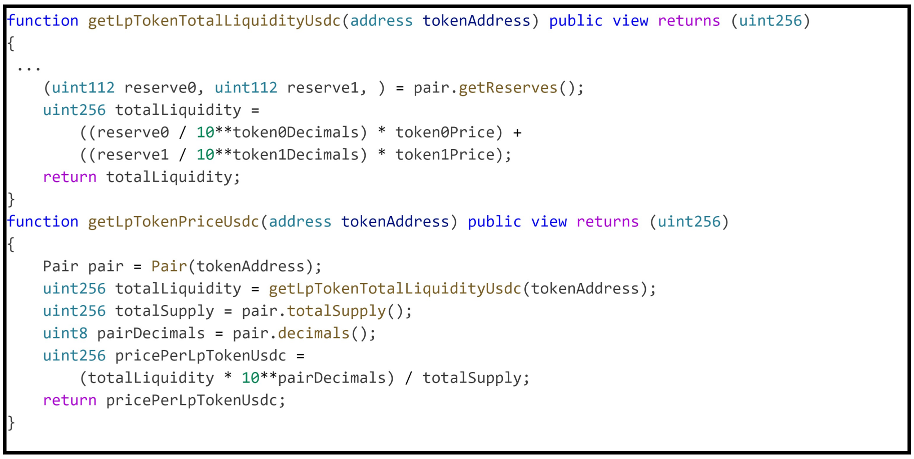
This study employs the following research methods: (1) Dataset Selection: The experimental datasets consist of three subsets—Web3Bugs, DefiHacks, and Top200—covering smart contracts of varying scales and complexities; (2) Evaluation Metrics: Metrics such as Precision, Recall, and F1-Score are utilized to comprehensively assess detection capabilities; (3) Experimental Procedure: Detection efficiency is optimized through vectorization techniques and iterative GPT analysis, with the framework's performance ultimately validated across multiple datasets.

4. EXPERIMENTS AND EVALUATION

This chapter will detail the evaluation methods, experimental parameters, and results of the VulnScan GPT framework in smart contract vulnerability detection. We analyze the performance of the VulnScan GPT framework on different vulnerabilities through experimental results to verify its effectiveness and reliability in vulnerability detection.

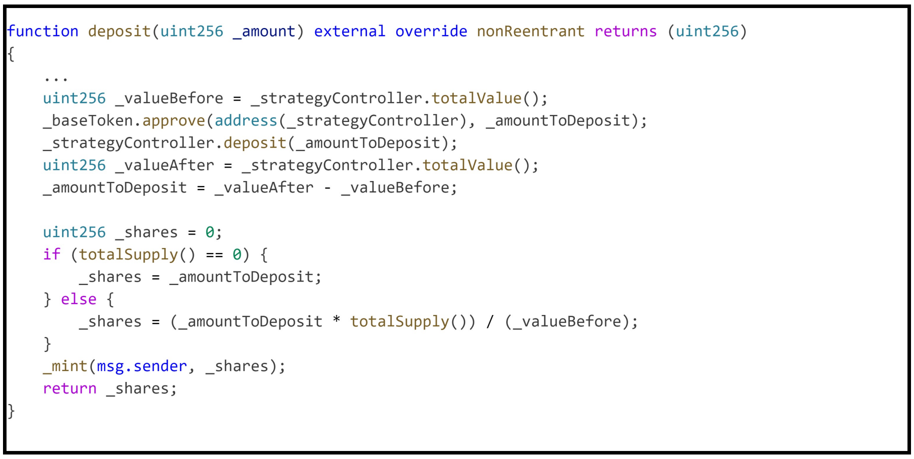
4.1. Dataset

We conducted an analysis on a total of 366 contracts. Among them, 20 project contracts were selected from the web3bugs dataset, and 66 contracts with identifiable vulnerabilities were selected from real attack cases recorded in DefiHacks[24]. For these 66 contracts, based on the attacks they have experienced, if there is no evidence of attackers profiting from automated market maker (AMM) vulnerabilities, we consider them to be free of such vulnerabilities. The remaining 280 contracts are from a dataset named Top200, which includes the top 200 smart contracts by market capitalization. This set includes 280 open-source contract projects from six major Ethereum-compatible chains. Since these projects have been deployed on the blockchain for a long time and are very popular, we assume they are free of vulnerabilities. The Web3Bugs dataset consists of smart contract samples sourced from real-world vulnerability detection platforms, characterized by diverse vulnerability types and comprehensive annotations. The DefiHacks dataset focuses primarily on recorded attack incidents, while the Top200 dataset reflects the security status of mainstream smart contracts in actual production environments. This dataset is mainly used to evaluate the false positive rate of VulnScan GPT when testing for AMM vulnerabilities. Since the method of obtaining AST trees through solcjs compilation is highly dependent on the solc version, we filtered out project contracts that involved multiple Solidity language versions within a single project. Additionally, all 20 project contracts selected from web3bugs were detected by GPTscan. Among these 20 contracts, nine were confirmed to have Automated Market Maker price manipulation vulnerabilities, and six were found to have Risk First Deposit (RFD) vulnerabilities. Based on the GPTScan results, we performed Automated Market Maker price manipulation vulnerability scans on 20 contracts and RFD vulnerability scans on ten contracts. In the DefiHacks dataset, we conducted targeted detection specifically for Automated Market Maker price manipulation vulnerabilities. Figure 5 and Figure 6 show examples of these two vulnerabilities, respectively.



```
function getLpTokenTotalLiquidityUsdc(address tokenAddress) public view returns (uint256)
{
 ...
    (uint112 reserve0, uint112 reserve1, ) = pair.getReserves();
    uint256 totalLiquidity =
        ((reserve0 / 10**token0Decimals) * token0Price) +
        ((reserve1 / 10**token1Decimals) * token1Price);
    return totalLiquidity;
}
function getLpTokenPriceUsdc(address tokenAddress) public view returns (uint256)
{
    Pair pair = Pair(tokenAddress);
    uint256 totalLiquidity = getLpTokenTotalLiquidityUsdc(tokenAddress);
    uint256 totalSupply = pair.totalSupply();
    uint8 pairDecimals = pair.decimals();
    uint256 pricePerLpTokenUsdc =
        (totalLiquidity * 10**pairDecimals) / totalSupply;
    return pricePerLpTokenUsdc;
}
```

Figure 5. AMM price manipulation from 2021-09-sushimiso.



```
function deposit(uint256 _amount) external override nonReentrant returns (uint256)
{
    ...
    uint256 _valueBefore = _strategyController.totalValue();
    _baseToken.approve(address(_strategyController), _amountToDeposit);
    _strategyController.deposit(_amountToDeposit);
    uint256 _valueAfter = _strategyController.totalValue();
    _amountToDeposit = _valueAfter - _valueBefore;

    uint256 _shares = 0;
    if (totalSupply() == 0) {
        _shares = _amountToDeposit;
    } else {
        _shares = (_amountToDeposit * totalSupply()) / (_valueBefore);
    }
    _mint(msg.sender, _shares);
    return _shares;
}
```

Figure 6. Risk first deposit from 100-LogicBug-Prepo.

4.2. Experimental parameters

In the experiment, we used the latest GPT-4 model provided by OpenAI (version: gpt-4-0125-preview). This model is renowned for its powerful language generation capabilities and wide range of applications. During the experiment, the model's Temperature parameter was set to 1, and the response_format parameter was set to type ”json_object” to ensure diversity, randomness, and format stability in the generated results. To achieve efficient text vectorization, we used OpenAI's text-embedding-ada-002 model. This model can convert text data into high-dimensional vectors, facilitating subsequent similarity calculations and other vector-based operations.

For the vector database, we chose the Chroma[25] local vector database. Chroma is widely praised for its efficient retrieval performance and flexible local storage solutions. In this experiment, the Chroma vector database is used to store and retrieve text vectors generated by the text-embedding-ada-002 model[3]. In the text processing phase, reasonable text segmentation can significantly improve the model's processing efficiency and the accuracy of the results. To this end, we used Langchain's Solidity-based text splitter[26] for function chunking. This text splitter can segment complex text content according to certain logic and rules, ensuring that the model maintains high contextual coherence when processing each text block.

4.3. Evaluation criteria

Evaluating the performance of a smart contract vulnerability detection system requires multiple metrics to comprehensively assess its detection capability and accuracy. The following are the main evaluation criteria we used:

TP: True Positive. The number of vulnerabilities correctly detected by the system. These vulnerabilities indeed exist in the actual contracts and have been successfully identified by the VulnScan GPT framework.

FP: False Positive, indicating the number of instances the system incorrectly reported as vulnerabilities. These are code segments misjudged by the system as having vulnerabilities, reflecting the system's false alarm rate.

FN: False Negative, the number of actual vulnerabilities that the system failed to detect. These vulnerabilities indeed exist in the smart contracts but were not identified by the VulnScan GPT framework.

Precision: Precision measures the accuracy of the system when detecting vulnerabilities, calculated as the ratio of true positives to the total number of detected vulnerabilities (including true positives and false positives). A high precision indicates that the system has a high accuracy when reporting vulnerabilities.

$$\text{Precision} = \frac{TP}{TP + FP} \tag{3}$$

Recall: Recall measures the proportion of actual vulnerabilities detected by the system out of the total number of actual vulnerabilities. A high recall indicates that the system can detect most of the actual existing vulnerabilities.

$$\text{Recall} = \frac{TP}{TP + FN} \tag{4}$$

F1 Score: To comprehensively consider precision and recall, we use the F1 score as a composite evaluation metric. The F1 score is the harmonic mean of precision and recall.

$$F1 = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} \tag{5}$$

4.4. Results

In this section, we introduce the advantages of VulnScan GPT from two aspects. First, we present the results in AMM price manipulation and RFD vulnerability detection. Second, we discuss the token savings achieved by using VulnScan GPT for detection. Unlike existing detection tools such as Mythril and Slither, the VulnScan GPT framework focuses on detecting complex semantic vulnerabilities, such as AMM price manipulation and initial risk deposit vulnerabilities. Traditional tools primarily rely on static analysis methods, which are limited to identifying structural errors or simple logical flaws and cannot detect complex behaviors that require deep contextual understanding. While the aforementioned types of vulnerabilities are beyond the detection scope of traditional tools, they have caused significant economic losses in practice (as evidenced by real attack cases in DeFi hacks). Therefore, a direct comparison with traditional tools is not meaningful. VulnScan GPT is more focused on expanding the detection scope and addressing the shortcomings of traditional methods. Table 1 and Table 2 illustrate the detection results, while Table 3 shows the token savings.

In the detection of AMM price manipulation vulnerabilities, VulnScan GPT achieved a precision of 58%, a recall of 88%, and an F1 score of 70% on the Web3bugs dataset. This is a slight improvement compared to the

Table 1. VulnScan GPT vulnerability detection results on different datasets

	Vulnerabilities (Dataset)	Precision	Recall	F1 Score
GPTScan	AMM(Web3bugs)	53%	100%	69%
VulnScan GPT	AMM(Web3bugs)	**58%**	88%	**70%**
GPTScan	RFD(Web3bugs)	50%	60%	55%
VulnScan GPT	RFD(Web3bugs)	**67%**	**67%**	**67%**
VulnScan GPT	AMM(DefiHacks)	70%	90%	79%
GPTScan	AMM(DefiHacks)	N/A	N/A	N/A

Table 2. Accuracy evaluation of VulnScan GPT on AMM vulnerabilities

	Dataset	TP	TN	FP	FN
VulnScan GPT	Web3bugs	7	7	5	1
VulnScan GPT	Top200	0	235	45	0
VulnScan GPT	DefiHacks	19	37	8	2

Table 3. A token usage comparison table for the detection of 66 project contracts in the DefiHacks dataset

Phase	Token usage
complete project contracts	977k
After Pruning	328k
With VlnScanGPT Framework	104k

69% F1 score of GPTScan. For the Defihacks dataset, VulnScan GPT achieved a precision of 70%, a recall of 90%, and an F1 score of 79%. This indicates that VulnScan GPT has made progress in reducing false positives while accurately detecting vulnerabilities in most cases (high recall). Due to technical reasons, we were unable to conduct GPTScan before the article's submission deadline, so the comparison is based on data released by GPTScan. The 100% recall rate of GPTScan is due to the fact that the selected contracts and the rules detected by GPTScan indeed involve related vulnerabilities, and we can also ensure that checks are not affected by compiler dependency issues. The same applies to the DefiHacks dataset, so there are no relevant detection results.

In the detection of RFD vulnerabilities, VulnScan GPT performed better in both precision and recall, achieving an F1 score of 67%, significantly higher than GPTScan's 55%. This indicates that VulnScan GPT has higher reliability and accuracy in detecting this type of vulnerability.

We found that on the defiHacks dataset and the Top200 dataset, the effect of VulnScanGPT is much higher than that of the Web3bugs dataset. This may be related to the fact that they are all token contracts compared to Web3bugs, and their logic is not as complicated as that of Web3bugs. Through our analysis, we found that certain vulnerability samples in the Web3bugs dataset exhibit higher logical complexity, leading to a significantly higher misclassification rate for these samples. This phenomenon may be attributed to the following factors: (1) the increased logical complexity poses challenges to the identification of AMM patterns; (2) the Web3bugs dataset contains more files and features significantly higher module complexity compared to the DefiHacks dataset. In contrast, the samples in the DefiHacks dataset generally exhibit lower logical complexity, enabling the model to capture their features more effectively and achieve better detection performance.

Complete project contracts: refer to submitting an entire contract directly to GPT for detection without any prior processing.

After Pruning: refers to the state after removing unnecessary third-party libraries (e.g., those from well-known libraries such as ”OpenZeppelin,” ”Uniswap,” and ”PancakeSwap”) and interface files before storing them in the vector database.

Table 3 demonstrates that our framework achieves an 89% reduction in token usage compared to directly analyzing the entire smart contract project using GPT. Additionally, following the pruning of non-essential files, our proposed VulnScan GPT framework further reduces token usage by 68% relative to the After Pruning method. This significant reduction in token consumption underscores the efficiency and cost-effectiveness of our approach in utilizing GPT for vulnerability detection. Based on the recorded token usage during the experiments, we estimate that under the current OpenAI API pricing standard ($0.01 per 1k tokens), VulnScan GPT achieves nearly 89% cost savings compared to direct detection methods. This significant cost reduction provides an economically viable solution for batch vulnerability detection in blockchain projects, particularly in large-scale, multi-project environments.

5. CONCLUSION

This study provides an in-depth evaluation of the efficiency of the VulnScan GPT framework in detecting vulnerabilities in smart contracts, with notable effectiveness in identifying critical issues such as AMM price manipulation and RFD vulnerabilities. The findings not only validate the framework's capability in vulnerability detection but also highlight its significant savings in token usage. However, the analysis revealed several challenges that require immediate attention, particularly the issue of search ambiguity when handling similar function signatures, which affects the detection accuracy. Specifically, first, the problem of function segmentation reveals the limitations of current text processing algorithms. Incorrect segmentation may lead to detection errors, indicating that automated text processing needs further optimization. We consider using regular expressions to build a script to segment solidity files specifically for smart contract writing habits. Secondly, the method of searching based solely on signature similarity has defects when facing similar function signatures, emphasizing the necessity of introducing more refined differentiation mechanisms in the vector database search phase to enhance the system's sensitivity to subtle differences. In addition, the inaccuracy of GPT in generating signatures further indicates that more effective prompt design and innovative strategies are needed to improve the reliability and accuracy of the generated results. We are temporarily considering adding the use of natural language descriptions to code blocks as a reference indexing method.

Looking ahead, future research will focus on addressing these technical challenges to improve the system's accuracy and efficiency. Specifically, regarding functional expansion of the framework, future efforts will concentrate on the following aspects: first, enhancing the framework's robustness against adversarial attacks and ensuring the integrity of vector database operations through data validation mechanisms and secure storage technologies[27]; second, improving real-time monitoring capabilities by incorporating dynamic data flow analysis and incremental detection techniques to meet the rapidly evolving security demands of blockchain technology; and third, integrating predictive models for historical vulnerability trend analysis to support the identification and prevention of potential risks. These enhancements will significantly improve the adaptability, scalability, and effectiveness of VulnScan GPT in blockchain security scenarios, providing more reliable tools for smart contract security assessments[19].

DECLARATIONS

Authors' contributions

Responsible for research design and experimental plan formulation; writing the main part of the paper: Wang L

Participated in data visualization and chart production; assisted in paper proofreading and language polishing: Zhao Y

Performed data collection and analysis; assisted in revising the paper: Wang K

Responsible for the evaluation and review of research ideas; led the overall review and quality control of papers: Zhang Z

Provided research funding support; participated in technical review of research results and paper editing: Zhang W

Availability of data and materials

Not applicable.

Financial support and sponsorship

None.

Conflicts of interest

All authors declared that there are no conflicts of interest.

Ethical approval and consent to participate

Not applicable.

Consent for publication

Not applicable.

Copyright

REFERENCES

1. Zhang Z, Zhang B, Xu W, Lin Z. Demystifying exploitable bugs in smart contracts. In: 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE). Melbourne, Australia: IEEE; May 14-20,2023. pp. 615–27. DOI
2. Wang Y, Kordi Y, Mishra S, et al. Self-instruct: aligning language models with self-generated instructions. *arXiv preprint arXiv:221210560* 2022. DOI
3. OpenAI; 2024. Available from https://openai.com/. [Last accessed on 25 Dec 2024]
4. Chen C, Su J, Chen J, et al. When chatgpt meets smart contract vulnerability detection: How far are we? *arXiv preprint arXiv:230905520* 2023. DOI
5. Han Y, Liu C, Wang P. A comprehensive survey on vector database: storage and retrieval technique, challenge. *arXiv preprint arXiv:231011703* 2023. DOI
6. Brent L, Jurisevic A, Kong M, et al. Vandal: A scalable security analysis framework for smart contracts. *arXiv preprint arXiv:180903981* 2018. DOI
7. Tsankov P, Dan A, Drachsler-Cohen D, et al. Securify: practical security analysis of smart contracts. In: Proceedings of the 2018 ACM SIGSAC conference on computer and communications security. Toronto Canada: Association for Computing Machinery, New York, NY, United States; October 15-19,2018. pp. 67–82. DOI
8. Feist J, Grieco G, Groce A. Slither: a static analysis framework for smart contracts. In: 2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB). Montreal, QC, Canada: IEEE; May 27-27,2019. pp. 8–15. DOI
9. Grieco G, Song W, Cygan A, Feist J, Groce A. Echidna: effective, usable, and fast fuzzing for smart contracts. In: Proceedings of the 29th ACM SIGSOFT international symposium on software testing and analysis. Virtual Event USA: Association for Computing Machinery, New York, NY, United States; July 18-22,2020. pp. 557–60. DOI
10. Jiang B, Liu Y, Chan WK. Contractfuzzer: fuzzing smart contracts for vulnerability detection. In: Proceedings of the 33rd ACM/IEEE international conference on automated software engineering. Montpellier France: Association for Computing Machinery, New York,NY,United States; September 3-7,2018. pp. 259–69. DOI
11. Wüstholz V, Christakis M. Harvey: a greybox fuzzer for smart contracts. In: Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. Virtual Event USA: Association for Computing Machinery, New York, NY, United States; November 8-13,2020. pp. 1398–409. DOI
12. Luu L, Chu DH, Olickel H, Saxena P, Hobor A. Making smart contracts smarter. In: Proceedings of the 2016 ACM SIGSAC conference on computer and communications security. Vienna Austria: Association for Computing Machinery, New York, NY, United States; October 24-28,2016. pp. 254–69. DOI
13. halmos;. 2024. Available from https://github.com/a16z/halmos. [Last accessed on 25 Dec 2024]
14. Mythril;. 2024. Available from https://github.com/Consensys/mythril. [Last accessed on 25 Dec 2024]
15. Kalra S, Goel S, Dhawan M, Sharma S. ZEUS: Analyzing Safety of Smart Contracts. In: Network and Distributed System Security Symposium; 2018. Available from: https://api.semanticscholar.org/CorpusID:3481056. [Last accessed on 25 Dec 2024]
16. Permenev A, Dimitrov D, Tsankov P, Drachsler-Cohen D, Vechev M. Verx: Safety verification of smart contracts. In: 2020 IEEE

symposium on security and privacy (SP). San Francisco, CA, USA: IEEE; May 18-21,2020. pp. 1661–77.

17. So S, Lee M, Park J, Lee H, Oh H. Verismart: A highly precise safety verifier for ethereum smart contracts. In: 2020 IEEE Symposium on Security and Privacy (SP). San Francisco, CA, USA: IEEE; May 18-21,2020. pp. 1678–94.
18. Rani P, Lamba R, Sachdeva RK, Kumar K, Iwendi C. A machine learning model for Alzheimer's disease prediction. *IET Cyber-Physical Systems: Theory & Applications* 2024. DOI
19. Taheri R. UNBUS: uncertainty-aware deep botnet detection system in presence of perturbed samples; 2022. DOI
20. Wang Z, Zhang L, Cao C, Liu P. The effectiveness of large language models (ChatGPT and CodeBERT) for security-oriented code analysis. *Available at SSRN 4567887* 2023. DOI
21. David I, Zhou L, Qin K, et al. Do you still need a manual smart contract audit? *arXiv preprint arXiv:230612338* 2023. DOI
22. Sun Y, Wu D, Xue Y, Liu H, Wang H, et al. Gptscan: Detecting logic vulnerabilities in smart contracts by combining gpt with program analysis. In: Proceedings of the IEEE/ACM 46th International Conference on Software Engineering. Association for Computing Machinery,New York,NY,United States; April 14-20,2024. pp. 1–13. DOI
23. Ma R, Jian Z, Chen G, Ma K, Chen Y. Rejection: A AST-based reentrancy vulnerability detection method. In: Trusted Computing and Information Security: 13th Chinese Conference, CTCIS 2019, Shanghai, China, October 24–27, 2019, Revised Selected Papers 13. Springer; 2020. pp. 58–71. DOI
24. DefiHacks;. 2024. Available from https://web3sec.notion.site/web3sec/I-m-SunSec-ddaa8bf9a985494dbaf70d698345b899. [Last accessed on 25 Dec 2024]
25. Chroma;. 2024. Available from https://docs.trychroma.com/. [Last accessed on 25 Dec 2024]
26. langchain;. 2024. Available from https://www.langchain.com/. [Last accessed on 25 Dec 2024]
27. Taheri R, Shojafar M, Arabikhan F, Gegov A. Unveiling vulnerabilities in deep learning-based malware detection: Differential privacy driven adversarial attacks. *Comput Secur* 2024;146:104035. DOI