ContractTinker: LLM-Empowered Vulnerability Repair for Real-World Smart Contracts

Che Wang
chewang@stu.pku.edu.cn
School of Computer Science
Peking University
Beijing, China

Jiashuo Zhang
zhangjiashuo@pku.edu.cn
School of Computer Science
Peking University
Beijing, China

Jianbo Gao*
gao@bjtu.edu.cn
Beijing Key Laboratory of Security and Privacy in Intelligent Transportation
Beijing Jiaotong University
Beijing, China

Libin Xia
lbxia@stu.pku.edu.cn
School of Computer Science
Peking University
Beijing, China

Zhi Guan
guan@pku.edu.cn
National Engineering Research Center For Software Engineering
Peking University
Beijing, China

Zhong Chen
zhongchen@pku.edu.cn
School of Computer Science
Peking University
Beijing, China

Abstract

Smart contracts are susceptible to being exploited by attackers, especially when facing real-world vulnerabilities. To mitigate this risk, developers often rely on third-party audit services to identify potential vulnerabilities before project deployment. Nevertheless, repairing the identified vulnerabilities is still complex and labor-intensive, particularly for developers lacking security expertise. Moreover, existing pattern-based repair tools mostly fail to address real-world vulnerabilities due to their lack of high-level semantic understanding. To fill this gap, we propose ContractTinker, a Large Language Models (LLMs)-empowered tool for real-world vulnerability repair. The key insight is our adoption of the Chain-of-Thought approach to break down the entire generation task into subtasks. Additionally, to reduce hallucination, we integrate program static analysis to guide the LLM. We evaluate ContractTinker on 48 high-risk vulnerabilities. The experimental results show that among the patches generated by ContractTinker, 23 (48%) are valid patches that fix the vulnerabilities, while 10 (21%) require only minor modifications. A video of ContractTinker is available at https://youtu.be/HWFVi-YHcPE.

CCS Concepts

• **Software and its engineering** → *Software development techniques*; • **Security and privacy** → *Software security engineering*.

Keywords

Program Repair, Smart Contract, Large Language Model

*Corresponding author.

ACM Reference Format:
Che Wang, Jiashuo Zhang, Jianbo Gao, Libin Xia, Zhi Guan, and Zhong Chen. 2024. ContractTinker: LLM-Empowered Vulnerability Repair for Real-World Smart Contracts. In *39th IEEE/ACM International Conference on Automated Software Engineering (ASE '24), October 27-November 1, 2024, Sacramento, CA, USA.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3691620.3695349

1 Introduction

Smart contracts are highly susceptible to various types of attacks due to their immutable nature and their close association with financial activities on blockchains. To prevent smart contracts from being attacked, developers of real-world applications often rely on third-party security audit services, *e.g.*, Code4Rena[3], to identify security vulnerabilities before deploying the contracts.

However, even with the information reported by the third-party audit service, vulnerability repair is a complex and labor-intensive task, especially for developers who may not be familiar with security attacks. Consequently, automated program repair (APR) has emerged as a highly popular research topic with immense practical value. It simplifies developers' tasks by automatically generating patches for specific types of vulnerabilities.

Several APR tools [6, 7] have been proposed to assist smart contract developers in implementing patches. These tools typically target vulnerabilities with clear low-level characteristics, such as *re-entrancy* and *integer overflow*, and generate patches based on predefined patterns. However, as shown in a recent empirical study [12], real-world vulnerabilities are often high-level functional bugs that are closely related to the business logic of the contract. Due to a lack of understanding of these high-level semantics, existing pattern-based methods may fail to effectively address real-world vulnerabilities.

To fill this gap, we developed ContractTinker, a vulnerability repair tool for real-world smart contracts empowered by LLMs. ContractTinker utilizes Large Language Models (LLMs) to understand high-level business logic of smart contracts, and extract structural information and vulnerability context from audit reports and smart

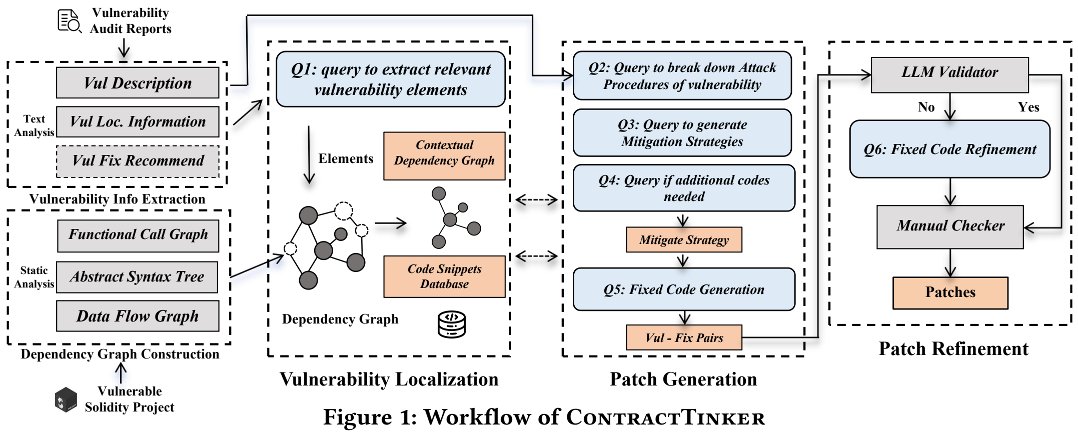

Vulnerability Audit Reports
Vul Description
Text Analysis
Vul Loc. Information
Vul Fix Recommend
Vulnerability Info Extraction
Functional Call Graph
Static Analysis
Abstract Syntax Tree
Data Flow Graph
Dependency Graph Construction
Vulnerable Solidity Project
Q1: query to extract relevant vulnerability elements
Elements
Contextual Dependency Graph
Code Snippets Database
Dependency Graph
Vulnerability Localization
Q2: Query to break down Attack Procedures of vulnerability
Q3: Query to generate Mitigation Strategies
Q4: Query if additional codes needed
Mitigate Strategy
Q5: Fixed Code Generation
Vul - Fix Pairs
Patch Generation
LLM Validator
No
Yes
Q6: Fixed Code Refinement
Manual Checker
Patches
Patch Refinement

Figure 1: Workflow of ContractTinker

contract code, respectively. To mitigate the impact of LLM's hallucination, we break down the entire complex task through simulating the behavior of smart contract expert, and infer it step by step as presented in section 2.3. Additionally, in each step we integrate structural information and program static analysis results to guide the LLMs in generating fine-grained patches. We summarize our contributions as follows:

- We proposed an LLMs-empowered tool, ContractTinker, for real-world smart contract vulnerability repair. We employed the Chain-of-Thought (CoT) mechanism to guide the LLM in mapping vulnerability descriptions in audit reports to the semantics of smart contracts and generating corresponding patches for these vulnerabilities.
- We incorporated static analysis, including dependency analysis and program slicing, to help LLMs localizing the vulnerability in complex real-world smart contract projects and improve the efficiency of patch generation.
- We open-sourced our tool and dataset at https://github.com/CheWang09/LLM4SMAPR. The preliminary evaluation on 48 real-world functional vulnerabilities demonstrates the effectiveness of ContractTinker. Of these, 23 (48%) are valid patches that fix the vulnerabilities, while 10 (21%) require only minor modifications.

2 Approach

The workflow of ContractTinker is illustrated in Figure 1. Firstly, users input the project and audit report. Then ContractTinker performs two steps: It analyzes the project to extract an entire dependency graph and extracts the valid structural information from reports. Upon obtaining the necessary elements, the tool performs vulnerability localization to filter out useless elements, and builds a contextual dependency graph (CDG) as well as program slices. During patch generation, we adopted the "chain of thought" concept, breaking down the task to implement patch inference step by step. Finally, the tool employs another independent LLM to evaluate the generated patch code and refine it if it does not fix the vulnerability. The final output is validated patches.

2.1 Dependency Analysis

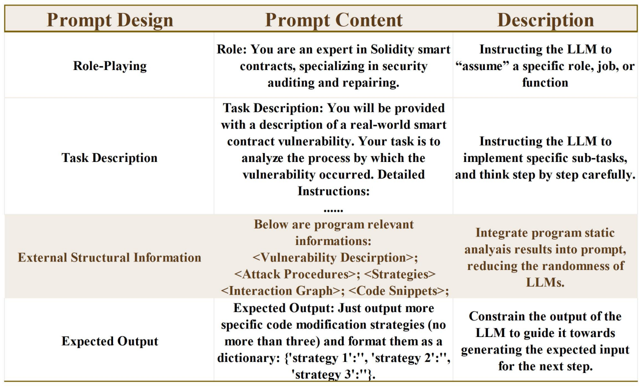
Since data and code are coupled in Solidity-based smart contracts, the interactions between functions and state variables, as well as among functions themselves, are crucial to understanding business logic errors. Therefore, to construct the dependency graph of the entire project, we defined three core elements included in dependency graph: function (F), state variables (V), and contracts (C). We define five types of edges in graph: $F-F$, $F-V$, $V-V$, $C-F$, $C-V$. Eventually, the dependency graph is defined as $DG = \{< V, E > | V \in \{F; V; C\},\ E \in \{[F, V] - [F, V]; C - [F, V]\}\}$.

When constructing a dependency graph, we mainly leverage three types of program analysis information: the Functional Call Graph, the Abstract Syntax Tree, and the Data Flow Graph. The Functional Call Graph is first utilized to build a base dependency graph of callee-caller between functions and contracts ($F-F$,$C-F$). Next, we extract the read-write relationship between functions/contracts and variables ($F-V$,$C-V$) that exist within the Abstract Syntax Tree. Finally, we extract data flow relationships among variables ($V-V$) from the Data Flow Graph to supply the dependency graph. These program structural information are utilized in Section 2.2 and Section 2.3 to locate vulnerabilities and generate patches, respectively.

2.2 Vulnerability Analysis & Localization

This section aims to locate vulnerability scope in project by combining extracted vulnerability information from text and dependency graph. First, we extract the vulnerability description from the audit report. Afterwards, we extract program information included in the description, such as vulnerability locations. This process is performed from two perspectives: First, it extracts explicitly pointed out vulnerable functions. Secondly, if the vulnerable function is not explicitly pointed out, we apply entity recognition [5] on the vulnerability description via LLMs, as shown in Figure 1 Q1, to extract relevant contracts, functions, and state variables.

Based on the above two preprocessed data sources, we employ program slicing, as used in previous studies [1], to localize the scope of vulnerable functions. We prioritize explicitly pointed-out functions as target functions to retrieve relevant elements in dependency graph. Then, we set recognized elements as seeds to retrieve relevant elements as supplement. At Last, we construct a contextual dependency graph (CDG), which is a pruned-graph of the dependency graph containing elements closely related to vulnerabilities. We create program slices consisting of all relevant code snippets of CDG.



Prompt Design	Prompt Content	Description
Role-Playing	Role: You are an expert in Solidity smart contracts, specializing in security auditing and repairing.	Instructing the LLM to “assume” a specific role, job, or function
Task Description	Task Description: You will be provided with a description of a real-world smart contract vulnerability. Your task is to analyze the process by which the vulnerability occurred. Detailed Instructions:	Instructing the LLM to implement specific sub-tasks, and think step by step carefully.
External Structural Information	Below are program relevant informations: <Vulnerability Descirption>; <Attack Procedures>; <Strategies> <Interaction Graph>; <Code Snippets>;	Integrate program static analyais results into prompt, reducing the randomness of LLMs.
Expected Output	Expected Output: Just output more specific code modification strategies (no more than three) and format them as a dictionary: {'strategy 1':'', 'strategy 2':'', 'strategy 3':''}.	Constrain the output of the LLM to guide it towards generating the expected input for the next step.

Figure 2: Prompt Design of ContractTinker

2.3 Patch Generation

Generally, a single patch generation task involves several internal inference steps, such as vulnerability localization, analysis, and patch generation. However, it is complex for the LLM to perform this reasoning, which could result in hallucination. Therefore, we employ the Chain of Thoughts approach as suggested by previous studies [9]. We simulated the behavior of smart contract experts and broke down patch generation into several reasoning steps as shown in Figure 1: vulnerability attack procedure analysis (Q2), mitigation strategies generation (Q3), and patch code generation (Q5). Q4 is a supplement to Q1. After learning the procedure of attacks, we aim to extract elements related to vulnerabilities from the dependency graph, which do not exist within the original textual descriptions.

Furthermore, in each step, we integrated structural text information and static analysis results into prompts to reduce LLM’s randomness. Specifically, we utilized vulnerability description and CDG in Q2. Q3 embeds attack procedures and CDG. Q4 is queried by combining attack procedures, CDG, and generated strategies. Afterwards, we supplemented CDG as well as the program slices. Eventually, we integrated CDG, program slices, and mitigation strategies into prompt of Q5 to generate Vul-Fix Pairs.

For all prompts, we designed a general template as shown in Figure 2. Its content mainly consists of several core parts: Role-Playing, Task Description, Expected Output, and External Information. We have open-sourced all prompt templates in the GitHub repository.

2.4 Patch Refinement

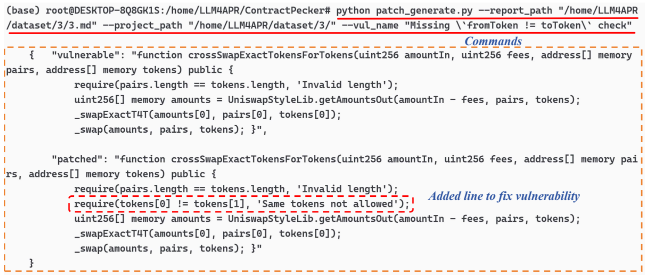
In this section, we introduce patch refinement which comprises of LLM Validator, Refinement, and Manual Checker. In particular, we adopt the concept of multi-agent debate [2] that another LLM is leveraged to judge whether the patches fixed the vulnerable code. If not, then the fixed code is inputted into Q6 in Figure 1 combined with supplemental recommendations. Finally, the fixed code is manually evaluated.

3 The ContractTinker Tool

3.1 Implementation

The ContractTinker consists of the following core modules:

- *Patch Generator* is the main module for generating patches. It implements vulnerability localization and constructs a pipeline for the entire generation task.



```
(base) root@DESKTOP-8Q8GK1S:/home/LLM4APR/ContractPecker# python patch_generate.py --report_path "/home/LLM4APR
/dataset/3/3.md" --project_path "/home/LLM4APR/dataset/3/" --vul_name "Missing \`fromToken != toToken\` check"
```

```
    {   "vulnerable": "function crossSwapExactTokensForTokens(uint256 amountIn, uint256 fees, address[] memory
pairs, address[] memory tokens) public {
            require(pairs.length == tokens.length, 'Invalid length');
            uint256[] memory amounts = UniswapStyleLib.getAmountsOut(amountIn - fees, pairs, tokens);
            _swapExactT4T(amounts[0], pairs[0], tokens[0]);
            _swap(amounts, pairs, tokens); }",

        "patched": "function crossSwapExactTokensForTokens(uint256 amountIn, uint256 fees, address[] memory pai
rs, address[] memory tokens) public {
            require(pairs.length == tokens.length, 'Invalid length');
            require(tokens[0] != tokens[1], 'Same tokens not allowed');
            uint256[] memory amounts = UniswapStyleLib.getAmountsOut(amountIn - fees, pairs, tokens);
            _swapExactT4T(amounts[0], pairs[0], tokens[0]);
            _swap(amounts, pairs, tokens); }"
    }
```

Figure 3: The Screenshot of ContractTinker

- *Document Parser* mainly focuses on parsing input audit reports, and it can be customized to support other types of audit formats.
- *Program Analyzer* is used to analyze the entire project, and construct dependency or data flow graphs to enhance the effectiveness of LLM-based CoT.
- *LLM Interface* is designed with several interfaces, enabling developers to utilize different LLMs with customized configurations.
- *Prompt utils* provides different prompt templates adopted for our sub-tasks. These prompt templates can be customized for specific sub-task settings.

Furthermore, in this paper, we utilized Slither[4] as primary program static analysis tool. We implemented a flexible interface to employ different LLMs, such as Llama, GPT-4, GPT-3.5, CodeT5. We used GPT-4 and GPT-3.5 as base model in our experiment.

3.2 Usage

We provide a command-line interface for developers to use our tool ContractTinker. The usage is simple, as presented in Figure 3. The detail of arguments is shown below to illustrate its usage:

- *—report_path* and *—project_path* are used to specify the exact report and project paths of the corresponding vulnerabilities.
- *—vul_name* specifies the vulnerability that needs to be repaired as included in the audit report.
- *—solc remaps* is also available for associating particular external dependency libraries (e.g., OpenZeppelin) to this project.
- *—output path* is used to store generated patches.

After preparing all inputs, the user can use the *CompilationChecker* module to test if the project can compile under the current environment. If the project cannot compile, the user needs to download the required packages as suggested by *CompilationChecker*. Then, the user runs *Python patch_generate.py* with the aforementioned arguments to generate patches for the specified vulnerability. Finally, the patch will be stored in the designated path. The orange box in Figure 3 illustrates an example of a vulnerability and its corresponding patch.

4 Preliminary Evaluation

4.1 Data Collection and Metrics

In this experiment, we collected 48 high-risk findings that contain both vulnerabilities and their fix recommendations from Code4Rena platform. The main metric to evaluate patches generated by the ContractTinker is accuracy. Additionally, we added strategy success rate, which is inspired by the Hit Rate (HR)[10] from recommendation systems, to evaluate the quality of generated strategies.

4.2 Evaluations

4.2.1 *Strategies Comparison.*

Before generating code patches, it is essential to demonstrate that ContractTinker can indeed provide correct mitigation strategies. We conduct an experiment, that uses the intermediate result to determine whether the fix recommendation matches one of the mitigation strategies generated by the ContractTinker. We adopt the strategy success rate metric to evaluate its effectiveness. Success Rate indicates the rate of single mitigation strategy generated by ContractTinker matches the fix recommendation. Top-3 Success Rate measures whether one of the top three mitigation strategies generated by ContractTinker matches the fix recommendation extracted from the audit report.

Results 1: *According to our evaluations, the* ***Success Rate*** *of ContractTinker with GPT-3.5 in generating strategy is* ***70.4%****, and the* ***Top3 Success Rate*** *in generating mitigation strategies is* ***91.6%****.*

4.2.2 *Patches Evaluation.*

In this section, we aim to evaluate the quality of the generated patches. Due to the lack of a finely labeled dataset, we employ three mechanisms to assess the accuracy of the patches: GPT-4 Check, Manual Check, and Compilation Check. Specifically, in compilation check, we compile the contract to examine if it can compile after incorporating the patches. In Validator check and manual check, we leverage GPT-4 and manual efforts to validate the functional correctness of the generated patches, respectively. Additionally, we categorize all patches into four classes: 1. Valid Patches that pass compilation, 2. Valid Patches that need minor modification, 3. Patches that need logic modification, and 4. Irreparable Patches.

Valid Patches that pass compilation represent patches that have passed all three check mechanisms. Valid Patches that need minor modification are those that have passed both the Validator Check and Manual Check but failed the Compilation Check. These patches contain minor issues, such as only included fixed code snippets directly related to the vulnerability functions or contained pseudo variable names that need to be replaced with state variable names defined in the contract. Thus, these patches require developers to make minor modifications without any logic changes.

Patches that need logic modification denote patches that have only partially implemented the necessary code. Since we input only relevant code snippets into our tool, it cannot fully understand the entire business logic of the projects. Therefore, the core logic code related to business functionalities is too complex for ContractTinker to implement correctly. Such as how to define time-weighted average price (TWAP) in a function when facing price manipulate attacks.

Irreparable Patches refer to patches where the vulnerability cannot be fixed by our tool. This can occur for several reasons: 1. The vulnerability description is too vague for ContractTinker to locate the bug and understand how the attack happens. 2. The business logic of the vulnerability code is too complex, leading ContractTinker to produce significant hallucinations. 3. Even if ContractTinker understands how the attack happens, it might not extract the correct target functions directly related to the vulnerability from description.

Results 2: *Based on our experiment, ContractTinker can generate* ***48%*** *of valid patches that* ***fix the vulnerabilities****.* ***21%*** *of patches require* ***only minor modifications****.*

5 RELATED WORK

The repair of smart contracts has been widely discussed over the past few years, while few of them can address real-world vulnerabilities. SGuard[6] relies on pre-defined patterns to fix four low-level vulnerabilities. Rodler [7] proposed EVMPatch, focusing on bytecode, and also only fixes low-level vulnerabilities. SCRepair[11] is a search based method limited by manually created unit tests. Smartfix[8] is the most recent proposed tool which uses statistical models to guide repair, but it is also limited to predefined vulnerability fixes. To our knowledge, these methods still focus on commonly known vulnerabilities and do not address real-world vulnerabilities that require comprehension of business logic or high-level semantics.

6 CONCLUSIONS

In this paper, we proposed ContractTinker, an LLM-empowered repair tool for real-world smart contract projects. We utilized the Chain-Of-Thought concept that breaks down generation task into sub-tasks, in which we integrated program static analysis, including dependency analysis and program slicing. This design enhanced the accuracy of the LLM by combining high-level semantic and static analysis. Our preliminary evaluation demonstrates the effectiveness of ContractTinker.

Acknowledgments

This work is supported by National Natural Science Foundation of China (62202011, 62172010).

References

[1] Sahar Badihi, Khaled Ahmed, Yi Li, and Julia Rubin. 2023. Responsibility in Context: On Applicability of Slicing in Semantic Regression Analysis. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE).*

[2] Chi-Min Chan, Weize Chen, Yusheng Su, Jianxuan Yu, Wei Xue, Shanghang Zhang, Jie Fu, and Zhiyuan Liu. 2023. Chateval: Towards better llm-based evaluators through multi-agent debate. *arXiv preprint arXiv:2308.07201* (2023).

[3] Code4Rena. [n. d.]. https://code4rena.com/reports Accessed: 2024-06-20.

[4] Josselin Feist, Gustavo Grieco, and et al. 2019. Slither: a static analysis framework for smart contracts. In *2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB).*

[5] Jing Li, Aixin Sun, Jianglei Han, and Chenliang Li. 2020. A survey on deep learning for named entity recognition. *IEEE transactions on knowledge and data engineering* (2020).

[6] Tai D Nguyen, Long H Pham, and et al. 2021. SGUARD: towards fixing vulnerable smart contracts automatically. In *2021 IEEE Symposium on Security and Privacy.*

[7] Michael Rodler, Wenting Li, and et al. 2021. EVMPatch: Timely and automated patching of ethereum smart contracts. In *30th usenix security symposium.*

[8] Sunbeom So and Hakjoo Oh. 2023. SmartFix: Fixing Vulnerable Smart Contracts by Accelerating Generate-and-Verify Repair using Statistical Models. In *Proc. 31st IEEE/ACM FSE.*

[9] Boshi Wang, Sewon Min, Xiang Deng, and et.al. 2023. Towards Understanding Chain-of-Thought Prompting: An Empirical Study of What Matters. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics.*

[10] Xin Wang, Yunhui Guo, and Congfu Xu. 2015. Recommendation algorithms for optimizing hit rate, user satisfaction and website revenue. In *Twenty-Fourth International Joint Conference on Artificial Intelligence.*

[11] Xiao Liang Yu. 2020. Smart contract repair. In *ACM Transactions on Software Engineering and Methodology (TOSEM).*

[12] Zhuo Zhang, Brian Zhang, and et al. 2023. Demystifying exploitable bugs in smart contracts. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE).*