Adversarial Generation Method for Smart Contract Fuzz Testing Seeds Guided by Chain-Based LLM

Jiaze Sun
Xi'an University of Posts and Telecommunications
Zhiqiang Yin
150298938450163.com

Xi'an University of Posts and Telecommunications
Hengshan Zhang
Xi'an University of Posts and Telecommunications
Xiang Chen
Nantong University
Wei Zheng
Northwestern Polytechnical University

Research Article

Keywords: Fuzz Testing, Smart Contracts, LLM, Seed Set Generation, Prompts

Posted Date: July 26th, 2024

DOI: https://doi.org/10.21203/rs.3.rs-4666159/v1

Additional Declarations: No competing interests reported.

Version of Record: A version of this preprint was published at Automated Software Engineering on December 31st, 2024. See the published version at https://doi.org/10.1007/s10515-024-00483-4.

Adversarial Generation Method for Smart Contract Fuzz Testing Seeds Guided by Chain-Based LLM

Jiaze Sun[1,2,3*], Zhiqiang Yin[1], Hengshan Zhang[1,2,3†], Xiang Chen[4†], Wei Zheng[5†]

[1*]School of Computer Science and Technology, Xi'an University of Posts and Telecommunications, Xi'an, 710121, China.
[2*]Shaanxi Key Laboratory of Network Data Analysis and Intelligent Processing(Xi'an University of Posts and Telecommunications), Xi'an, 710121, China.
[3*]Xi'an Key Laboratory of Big Data and Intelligent Computing(Xi'an University of Posts and Telecommunications), Xi'an, 710121,China.
[4]School of Information Science and Technology, Nantong University, Nantong, 226019, China.
[5]School of Software, Northwestern Polytechnical University, Xi'an, 710072, China.

*Corresponding author(s). E-mail(s): sunjiaze@xupt.edu.cn;
Contributing authors: 15029893845@163.com; zhanghs@xupt.edu.cn; xchencs@ntu.edu.cn; wzheng@nwpu.edu.cn;
†These authors contributed equally to this work.

Abstract

With the rapid development of smart contract technology and the continuous expansion of blockchain application scenarios, the security issues of smart contracts have garnered significant attention. However, traditional fuzz testing typically relies on randomly generated initial seed sets. This random generation method fails to understand the semantics of smart contracts, resulting in insufficient seed coverage. Additionally, traditional fuzz testing often ignores the syntax and semantic constraints within smart contracts, leading to the generation of seeds that may not conform to the syntactic rules of the contracts and may even include logic that violates contract semantics, thereby reducing the efficiency of

fuzz testing. To address these challenges, we propose a method for adversarial generation for smart contract fuzz testing seeds guided by Chain-Based LLM, leveraging the deep semantic understanding capabilities of LLM to assist in seed set generation. Firstly, we propose a method that utilizes Chain-Based prompts to request LLM to generate fuzz testing seeds, breaking down the LLM tasks into multiple steps to gradually guide the LLM in generating high-coverage seed sets. Secondly, by establishing adversarial roles for the LLM, we guide the LLM to autonomously generate and optimize seed sets, producing high-coverage initial seed sets for the program under test. To evaluate the effectiveness of the proposed method, 160 smart contracts were crawled from Etherscan for experimental purposes. Results indicate that using Chain-Based prompts to request LLM to generate fuzz testing seed sets improved instruction coverage by 2.63% compared to single-step requests. The method of generating seed sets by establishing adversarial roles for the LLM reduced the time to reach maximum instruction coverage from 60 seconds to approximately 30 seconds compared to single-role methods. Additionally, the seed sets generated by the proposed method can directly trigger simple types of vulnerabilities (e.g., timestamp dependency and block number dependency vulnerabilities), with instruction coverage improvements of 3.3% and 3.8%, respectively.

Keywords: Fuzz Testing, Smart Contracts, LLM, Seed Set Generation, Prompts

1 Introduction

A smart contract is an automated computing program that runs on a blockchain. It executes transactions and operations based on predefined rules and conditions in the contract terms, eliminating the need for third-party trust and achieving decentralized programmable functionality. Smart contracts can automatically execute contract terms, manage digital assets, and conduct financial transactions, providing a secure, transparent, and tamper-proof digital contract solution. Due to the immutability and decentralization of blockchain and smart contracts, once a smart contract is deployed on the blockchain, it cannot be modified. While immutability ensures trust and transparency, it also means that any vulnerabilities or errors in the code cannot be easily corrected without deploying a new version of the contract. However, frequent security incidents in smart contracts not only cause significant economic losses but also threaten the stability and trustworthiness of the blockchain ecosystem.

The security of smart contracts has received significant attention(Atzei et al. 2017). The complexity and specificity of smart contracts make it challenging for traditional software testing techniques to fully cover potential vulnerabilities and security risks. Fuzz testing is a method of automated test case generation aimed at discovering software vulnerabilities and defects by injecting large amounts of abnormal or randomly generated data into the system under test. Fuzz testing primarily focuses on triggering abnormal behaviors in the tested program to reveal potential security vulnerabilities, runtime errors, and stability issues. Fuzz testing technology has become one of the most effective methods in software testing and vulnerability discovery, capable of

identifying defects that cause system malfunctions. Consequently, smart contract fuzz testing, as an effective security testing method, is being widely applied in the security assessment and vulnerability discovery of smart contracts. A common fuzz testing method is mutation-based fuzz testing, where an initial set of seed inputs is created, typically consisting of valid input data, and then these seed data are repeatedly and slightly randomly modified (mutated) to generate a large number of new test cases.

However, traditional smart contract fuzz testing methods have some limitations(Jiang et al. 2018;Liu at al. 2018;Wustholz at al. 2020) that may affect their effectiveness and reliability in smart contract security assessment and vulnerability discovery. Firstly, the coverage of the initial seed set in traditional fuzz testing is insufficient. Traditional fuzz testing generally relies on randomly generated initial seed sets, which cannot fully cover the complex logic and edge cases of smart contracts, leading to insufficient seed coverage. Smart contracts often contain complex conditional branches, loop structures, data interactions, and complex logic involving blockchain interactions. Traditional random generation methods find it difficult to produce representative seed sets, thereby limiting the effectiveness of fuzz testing. Secondly, traditional fuzz testing often ignores the syntactic and semantic constraints within smart contracts. The syntactic and semantic rules of smart contracts are crucial for their correctness and security, but traditional fuzz testing methods usually overlook these rules, resulting in seeds that may not comply with the syntactic rules of the contracts and may even include logic that contradicts the contract semantics. In such cases, even if seeds are generated, it is difficult to effectively trigger potential vulnerabilities in the smart contract, thereby reducing the effectiveness of fuzz testing.

To address the issues of insufficient coverage of generated seeds due to traditional fuzz testing methods' inability to understand the semantics of smart contracts, as well as the ineffectiveness of seeds generated without considering syntactic and semantic constraints, which affects fuzz testing efficiency, this paper proposes an innovative method using Large Language Model (LLM) to generate initial seed sets for smart contract fuzz testing: Adversarial Generation Method for Smart Contract Fuzz Testing Seeds Guided by Chain-Based LLM. Unlike traditional fuzz testing methods, this approach involves an in-depth analysis of the source code of the smart contract under test by the LLM before executing the fuzz test, aiming to generate a high-coverage test case set.

Specifically, to guide the LLM in generating and optimizing the test case set, we design an automatic prompts construction method. The task of generating fuzzing test cases for smart contracts is divided into four steps. These prompts, compared to single-step requests to the LLM, help the LLM understand the functionality and structure of the smart contract, thereby generating more targeted test cases, improving the coverage and quality of the test case set. Secondly, to further enhance the quality and usability of the test case set, we adopt an adversarial strategy by assigning two key roles to the LLM: generator and evaluator. The generator is responsible for generating an initial test case set based on the LLM's language understanding and generation capabilities. The evaluator is responsible for optimizing the generated test cases set. This adversarial mechanism ensures that the generated test case set meets the syntactic, semantic, and logical requirements of the smart contract, thereby improving the

effectiveness and accuracy of the test case set. Finally, the test case set is transformed into a seed set for the target fuzz testing system and loaded as the initial seed set into the fuzz testing system.

In the experimental analysis, we tested 160 smart contracts crawled from Etherscan and compared the effectiveness of the proposed method with that of traditional fuzz testing methods using randomly generated initial seed sets. The experimental results show that the method of using Chain-Based prompts to request the LLM to generate the fuzz testing seed set improves instruction coverage by 2.63% compared to the method of single-step requests to generate the seed set. The method of establishing adversarial roles for the LLM to generate the seed set reduces the time to reach maximum instruction coverage by approximately 25 seconds compared to the single-role method. Furthermore, the seed sets generated by the proposed method can directly trigger simple types of vulnerabilities (such as timestamp dependence vulnerabilities and block number dependence vulnerabilities), with instruction coverage improvements of 3.3% and 3.8%, respectively. This indicates that the proposed method can effectively improve the efficiency and coverage of smart contract fuzz testing, providing a more reliable guarantee for the security and reliability of smart contracts.

(1)To address the initial seed generation problem for smart contract fuzz testing, the paper proposes adversarial generation method for smart contract fuzz testing seeds guided by Chain-Based LLM. To guide the LLM in generating text that meets specific requirements, a Chain-Based prompts method is proposed. To address the issues of insufficient coverage and low efficiency of the seed set generated by traditional random initial seed generation fuzz testing methods, an adversarial LLM is proposed, utilizing the LLM's deep semantic understanding capabilities to autonomously generate and optimize the initial seed set for fuzz testing.

(2)Three datasets (B1, B2, and B3) were selected as experimental subjects, with B1 and B2 consisting of 30 and 100 smart contracts crawled from the Ethereum blockchain, respectively, and B3 comprising a dataset of mainstream vulnerable smart contracts. The effectiveness of the proposed method was evaluated in terms of instruction coverage and testing time consumption. Multiple sets of experiments were conducted to validate that the proposed method shortens testing time and improves instruction coverage compared to traditional methods.

(3)To facilitate the expansion of our research work, we have open-sourced the code at: https://github.com/2590438469/LLM-Chain.git.

The remaining part of this paper is organized as follows.Section 1 introduces smart contract fuzz testing and related methods and research on large language models,Section 2 describes the design method of Chain-Based prompts,Section 3 introduces the adversarial LLM method for generating seeds, Section 4 validates the effectiveness of the proposed method through experimental analysis, and section 5 summarizes the entire paper.

2 Background

2.1 Smart Contract Fuzz Testing

In the field of software development, ensuring software quality and security is crucial. Fuzz testing, as a powerful automated testing technique, has garnered significant attention. Fuzz testing detects vulnerabilities and errors in software by inputting abnormal, random, or invalid data into the software. The core idea of fuzz testing is to automatically or semi-automatically generate random data based on certain rules, input the generated data into the program, and monitor the program for any anomalies to discover potential errors, such as memory leaks, system crashes, and unhandled exceptions. Once a fuzz testing generator is initiated and running, it autonomously searches for vulnerabilities without the need for manual intervention, which is highly effective in uncovering defects that traditional testing methods or manual audits cannot detect. As shown in Fig. 1, fuzz testing includes several basic steps: determining the system under test, using a generator and mutator to create system inputs, loading the inputs into the target program, monitoring the target program, detecting security issues, and outputting crash logs.

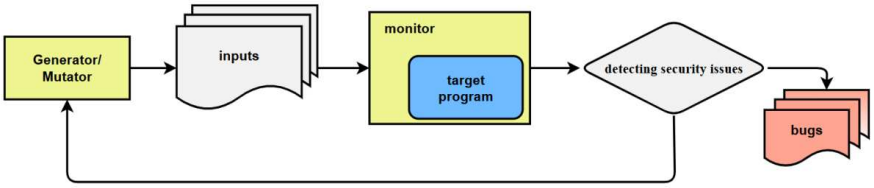

Fig. 1 Fuzz Testing Flowchart

A smart contract is an automated contract based on blockchain technology that exists on the blockchain in the form of code, executing the logic and functions stipulated in the contract. The advent of smart contracts has made the execution of traditional contracts more transparent, efficient, and trustworthy, as they can automatically execute contract terms without the intervention of intermediaries. The implementation of smart contracts relies on the characteristics of blockchain technology, primarily decentralization, immutability, and intelligence. Decentralization ensures that the execution of the contract does not depend on a centralized control entity but is verified and executed by multiple nodes in the blockchain network. Immutability ensures that the execution results of the contract cannot be tampered with or revoked, with all transactions and state changes recorded on the blockchain. Intelligence refers to the ability to write code with smart logic and automatic execution functions, enabling the implementation of various complex business logics. With the continuous development and popularization of blockchain technology, the application scenarios of smart contracts are constantly expanding. More and more enterprises and organizations are beginning to apply smart contracts in practical business to improve efficiency, reduce costs, and enhance security.

In the field of smart contract fuzz testing, Jiang et al. proposed the first smart contract fuzz testing framework, ContractFuzzer(Jiang et al. 2018). Liu et al. proposed ReGuard(Liu et al. 2018), a fuzz-based analyzer that can automatically detect reentrancy vulnerabilities in Ethereum smart contracts. Hu et al. proposed Harvey(Wustolz et al. 2020), a gray-box fuzz testing tool for smart contracts that enhances standard gray-box fuzz testing to better predict new paths or reveal vulnerabilities in smart contracts. Shou et al. introduced a new snapshot-based fuzz testing tool, Ityfuzz(Shou et al. 2023), for testing smart contracts. Sakhrawi et al. proposed an automated regression testing model based on ontology and the COSMIC functional size measurement method(Sakhrawi et al. 2024). Miah et al. proposed a model based on Hierarchical Graph Attention Network (HGAT)(Miah et al. 2024) by designing a specific set of metrics and applying regression and classification models to predict defects in smart contracts, also studying the impact of class imbalance on model performance. However, traditional fuzz testing seed generation methods generate inputs based on Application Binary Interface (ABI) analysis. ABI is the binary interface between an application and the operating system or library, defining the underlying details of function calls, parameter passing, and data types to ensure interoperability between code generated by different compilers. Smart contracts typically consist of complex business logic, and generating inputs based on ABI analysis does not deeply understand the actual logic and business meaning of the smart contract. Additionally, the generated inputs cannot cover all edge cases and special cases, leading to insufficient fuzz testing coverage. Therefore, this paper introduces Large Language Model (LLM) to assist in generating initial seed sets by understanding the semantics of smart contracts, addressing the shortcomings of traditional fuzz testing seed generation.

2.2 LLM-Based Smart Contract Fuzz Testing

Large language model (LLM) refer to language generation models with a massive number of parameters trained using deep learning techniques. These models, by learning from extensive text data, can generate semantically coherent and logically reasonable text, demonstrating powerful performance in various natural language processing (NLP) tasks. The advent of LLMs has profoundly impacted the field of NLP, significantly improving task performance and promoting research and applications in areas such as natural language understanding, generation, and dialogue systems. Concurrently, LLMs have sparked discussions and research regarding their capabilities, data usage, and privacy protection.

OpenAI's ChatGPT(Wu et al. 2023), currently at version GPT-4.0, is the most comprehensive model in terms of metrics. The development of ChatGPT can be divided into several key stages, progressing from CodeBERT, GraphCodeBERT, and CodeT5 to the current ChatGPT, with each stage featuring significant technical advancements and innovations. CodeBERT(Feng et al. 2017) is a bidirectional encoder representation model specifically designed for handling mixed texts of programming and natural languages, useful for tasks such as code search, code documentation generation, and other code understanding tasks. It was pre-trained on a large corpus of code and natural language descriptions, enabling it to comprehend and generate code-related texts. GraphCodeBERT(Guo et al. 2020) improves upon CodeBERT by

incorporating graph neural networks (GNN) to handle the graph structures of code. CodeT5(Wang et al. 2021) is a code generation and understanding model based on the T5 (Text-To-Text Transfer Transformer) model, unifying all tasks (e.g., code completion, code translation, code repair) into a text-to-text format. ChatGPT, based on the GPT (Generative Pre-trained Transformer) architecture, boasts strong natural language understanding and generation capabilities through pre-training on vast amounts of text data.

Additionally, Anthropic's model Claude(Enis et al. 2024), currently at version 3.0, features a stable context length of 200K tokens. Google's Gemini(Mihalache et al. 2024) model supports the longest context of 10M tokens. Mistral AI's closed-source model, Mistral Large(Zolkepli et al. 2024), released in February 2024, has a context length of 32K tokens. In the image generation domain, Midjourney Research Lab's AI program can perform text-to-image and image-to-image generation(Radhakrishnan et al. 2023). OpenAI's AI image generator DALL·E 3(Borji et al. 2022) offers greater diversity and creativity, generating various forms of images based on text descriptions.

In the realm of combining LLM with traditional fuzz testing, Xia et al. introduced a fuzz testing tool named Fuzz4All(Xia et al. 2024), the first universal fuzz testing tool capable of testing multiple input languages and their various characteristics. Hu et al. proposed CHATFUZZ(Hu et al. 2023), a gray-box fuzz testing tool enhanced by generative AI. Ackerman et al. proposed a natural language processing-based method to generate fuzz testing instances by exploiting ambiguities in format specifications(Ackerman et al. 2023). Deng et al. introduced TitanFuzz(Deng et al. 2023), the first method to directly use large language models to generate input programs for fuzz testing deep learning libraries. Building on this, Deng et al. proposed FuzzGPT(Deng et al. 2023), enabling LLM to synthesize abnormal programs for fuzz testing. Zhang et al. suggested a method using LLM for effective fuzz driver generation(Zhang et al. 2023). Lemieux et al. developed CodaMosa(Lemieux et al. 2023), which requests Codex to provide example test cases for uncovered functions when fuzz testing coverage halts. Gu et al. proposed an LLM-based code generation method(Gu et al. 2023), enhancing the quality and quantity of generated code through filtering and seed scheduling strategies. Qiu et al. introduced CHEMFUZZ(Qiu et al. 2017), using LLM to mutate and generate syntactically and semantically valid input files, generating new test cases from seed inputs to improve input validity.

Prompt design is considered crucial in optimizing LLM performance. Ma et al. proposed and evaluated a framework named TrustLLM(Ma et al. 2024), combining fine-tuning and LLM-based proxies for smart contract auditing. Meng et al. suggested CHATAFL(Meng et al. 2024), a protocol-guided fuzz testing engine using LLM, constructing message type grammars and interacting with LLM to generate and mutate message sequences. In the field of enhancing LLM prompts, Wei et al. introduced the Chain-of-Thought (CoT) strategy(Wei et al. 2022), aiming to enhance prompt effectiveness by guiding LLM to generate a series of intermediate steps before providing the final answer, thereby improving performance. Hu et al. developed GPTLENS(Hu et al. 2023), a method for using LLM to identify vulnerabilities in smart contracts.

LLM possess deep semantic understanding capabilities, enabling them to better grasp the meaning and contextual information of natural language. Therefore,

this paper posits that seeds generated based on LLM are typically more accurate in reflecting real-world usage scenarios, thereby improving seed quality and effectiveness. Moreover, LLM-generated seeds can cover more functional paths. By understanding semantic and contextual information, LLM can generate more comprehensive seeds. Additionally, due to LLM's understanding of semantic and contextual information, the generated seeds are often able to consider different aspects of the system more thoroughly, enhancing seed effectiveness. Hence, this paper proposes a method of using LLM to generate initial seed sets for fuzz testing, addressing the shortcomings of traditional fuzz testing systems' random seed generation, which leads to insufficient coverage and low fuzz testing efficiency.

3 Chain-Based Prompts

The core work of the article is to use LLM to generate an initial seed set before intelligent contract fuzz testing. Therefore, how to design prompts to fully tap into the potential of LLM becomes crucial. In this section, a method of Chain-Based Prompts is proposed to guide LLM to produce the expected results. Section 2.1 provides an overview of how Chain-Based Prompts are used to request LLM to address general issues. Section 2.2 applies Chain-Based Prompts in test case generation and elaborates on all steps involved in generating test cases using Chain-Based Prompts. In Section 4.1, experimental evidence is provided to demonstrate the effectiveness of the proposed method.

3.1 Overview of Chain-Based Prompts

As one of the prominent technologies in the field of artificial intelligence, LLM possess remarkable language generation capabilities. However, guiding these models to generate text that meets specific requirements presents a challenging task. In practical applications of LLM, it is customary to start with a simple question posed to the model and iteratively refine or supplement the prompts based on the responses until the desired answer is obtained(Liu et al. 2023). This approach has been applied to the 540-billion-parameter PaLM model(Chowdhery et al. 2023), where multi-step prompting significantly enhanced performance in several benchmark reasoning tasks compared to single-step prompting. For instance, in the GSM8K benchmark (17.9%?58.1%), which comprises 8000 stories, multi-step prompting demonstrated notable improvement.

Therefore, in the design process of multi-step prompts, this article proposes an iterative approach termed "Chain-Based Prompts" This method involves continuously optimizing and refining prompts while incorporating additional information to ensure the generation of optimal text by LLM. The key to designing chain prompts lies not only in providing a series of instructions or keywords to the model but also in fostering a dynamic and interactive process. The design of chain prompts needs to consider the complexity of the task, the language understanding capabilities of the LLM model, and user feedback.

This section aims to explore how to design universal Chain-Based Prompts to guide LLM in achieving optimal results for various tasks. Chain-Based Prompts involve gradually providing a series of prompts or instructions to guide LLM in generating text

on specific topics or in certain styles. Compared to traditional single prompts, Chain-Based Prompts offer greater flexibility and diversity, enabling more precise guidance for LLM to produce the desired text output.

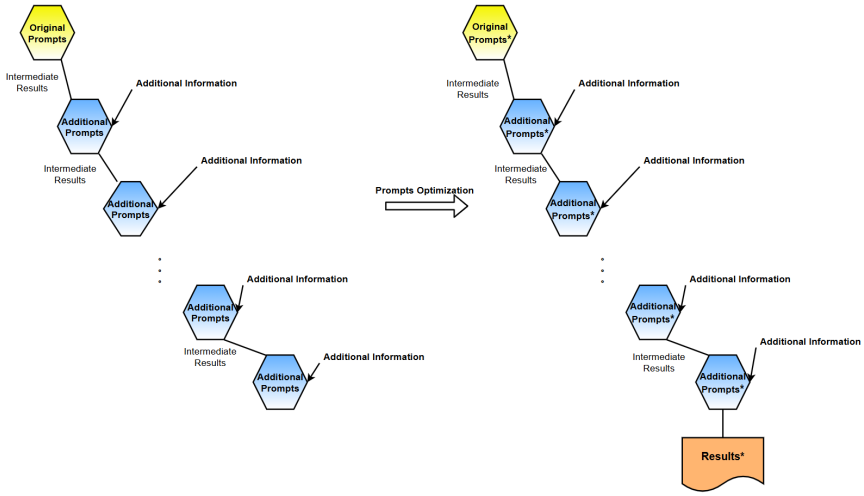

Fig. 2 Chain-Based Prompts to Guide LLM Text Generation

To guide LLM in task execution, this article adopts two main steps in designing Chain-Based Prompts, as illustrated in Fig. 2:

Prompt Description: Through in-depth analysis of task requirements, basic prompts are designed. These basic prompts aim to cover key information of the task as comprehensively as possible and effectively guide LLM in generating initial text results. The emphasis in this stage is to ensure that the provided prompts can stimulate the language generation capabilities of LLM to produce text content that meets expectations to a certain extent.

Multi-Step Optimization: However, solely relying on basic prompts often fails to fully meet the requirements of complex tasks. Therefore, this article introduces a multi-step optimization stage. In this stage, additional information is continuously added to optimize and update the results based on LLM feedback and user understanding. Specifically, the first step involves supplementing and adjusting the information in the prompt chain based on the LLM's generation results and user feedback to better align with task requirements. The second step utilizes the precision and accuracy of LLM in text to request LLM to optimize all previous prompts, ensuring the logic and coherence of the entire prompt chain. As shown in Fig. 2, the optimized prompts have "*" added to indicate the optimization.

Through the iterative process of these two steps, LLM can be effectively guided to generate optimal text results, thereby improving the accuracy and quality of text generation. This dynamic and interactive design approach not only fully leverages the language generation capabilities of LLM but also better meets the requirements and expectations of tasks.

3.2 Chain-Based Prompts from Text to Test Cases

In fuzz testing, seed mutation serves as a common strategy for generating new test cases. It relies on existing test cases and directly impacts the diversity and coverage of fuzz testing. Seed mutation is based on existing test cases to generate slightly different new test cases to ensure the diversity of test cases, thereby improving the coverage of fuzz testing and more comprehensively exploring potential errors in the program. Therefore, the generation of initial seed sets is crucial.

Traditional fuzz testing typically adopts random generation methods to generate initial seed sets(Xia et al. 2024). However, the random generation of seed methods lacks analysis of code semantics and application scenarios, resulting in insufficient coverage of seeds. In contrast, LLMs involve millions to billions of code snippets during pre-training, covering various programming languages, application scenarios, and domains. Through such large-scale code pre-training, models can fully learn programming knowledge and patterns, better understand the meaning and context of the code, and associate code snippets with comments or documentation.

Table 1 Chain-Based Prompts

No	Prompt Type	Text to Test Case Generation Task
P1	Task Prompt	Generate a high coverage test case set based on the source code
P2	Context Prompt	Provide code comments and example test cases
P3	Review Prompt	Check test case format and standardize parameter types
P4	Optimization Prompt	Optimize the prompts

In this section, based on the Chain-Based Prompts method proposed in Section 2.1, this paper proposes a design method for generating test cases using Chain-Based Prompts: guiding LLM step by step to generate a high-quality test case set. Specifically, the process of utilizing LLM to generate test cases is divided into four steps, as shown in Table 1: task prompt (P1), context prompt (P2), review prompt (P3), and optimization prompt (P4). Below is a detailed explanation of each step:

Task Prompt (P1): In this step, an initial prompt is designed to guide the generation task of the test case set. The prompt requires generating a sufficient number of test cases to cover all functions, branches, and statements within the functions. The specific prompt is as follows: "Contract Source Code Here is my smart contract, please generate a sufficient number of test cases to cover all functions, branches, and statements."

Context Prompt (P2): In this step, it was found that the generated test cases were disorderly formatted and did not well represent the content of the contract. Therefore, a context prompt is added to provide code explanations and example test cases for reference. The specific prompt is as follows: "Source Code Explanation, we need you to generate test cases according to the following template: JSON format."

Review Prompt (P3): Although the requirements are basically met in the previous two steps, there are still some issues with the format and parameter types of some generated test cases. Therefore, an review prompt is designed to request a recheck

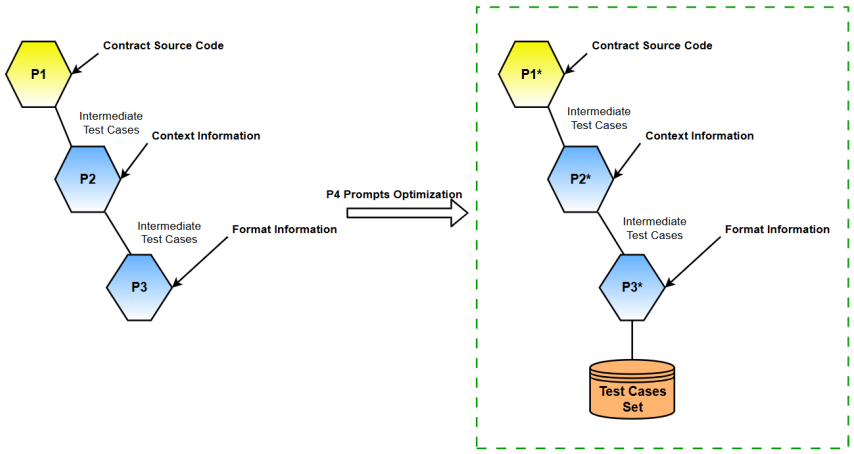

Fig. 3 LLM Chain-Based Prompt Test Case Generation Flowchart

of the generated test case format and to point out specific issues, such as parameter type mismatches. The specific prompt is as follows: "Please double-check the format of the generated test cases based on the template provided by me; the address type data should be a 40-character hexadecimal string like format example."

Optimization Prompt (P4): Due to the inaccuracies and lack of coherence in the existing manually designed prompts, it is necessary to request further optimization of all these manually designed prompts by LLM. Therefore, all prompts (P1, P2, P3) are provided to LLM, requesting it to improve the vocabulary and sentences. The prompt is as follows: "Original Prompt, here is a prompt provided by me, please improve the vocabulary and sentences." This step can utilize LLM's language generation ability to enhance the quality of prompts, making them more accurate and fluent, thereby improving the overall performance and effectiveness of the system.

As shown in Fig. 3, prompts P1, P2, and P3 are designed sequentially, and the information required for each step is provided. After determining the composition of the chain prompts, the prompts from the previous three steps are provided to LLM for further optimization, ultimately obtaining the model as shown on the right side of Fig. 2. Providing LLM with information such as source code, context, and format along the chain will yield the test case set.

4 Adversarial LLM

To fully leverage the semantic understanding and intelligent generation capabilities of LLM, this paper proposes a method to establish Adversarial LLM for generating test cases in fuzz testing. Two roles are defined for the same LLM: generator and evaluator. The generator is responsible for generating initial test cases for smart contracts based on the contract source code, while the evaluator further optimizes the generated test cases. The method of establishing adversarial LLM is derived from the following analysis:

(1)Adversarial methods have been proven to play a crucial role in improving the robustness and security of models in computer science and machine learning fields.

(2)The test cases actually generated by LLM have redundant issues, which lead to longer test times.

(3)The coverage of test cases generated by LLM in a single iteration is insufficient, requiring guidance for secondary optimization to fully exploit the potential of LLM.

The goal of generating more test cases is in conflict with the goal of optimizing and reducing test cases in the current paradigm of test case generation. Therefore, this paper divides a single test case generation into two adversarial stages: the generation stage and the evaluation stage. The idea of dividing one stage into multiple stages is also applied in deep learning(Goodfellow et al. 2014). By training two neural network models, referred to as the generator and the discriminator, these two models mutually learn through adversarial processes to generate high-quality data samples.

These test cases serve as the starting point for fuzz testing, covering different code paths and execution scenarios of smart contracts. Unlike the producer, the judge plays a role in optimizing the generated test cases. The judge's task is to further refine and optimize the test cases generated by the producer to improve the coverage and quality of the test cases.

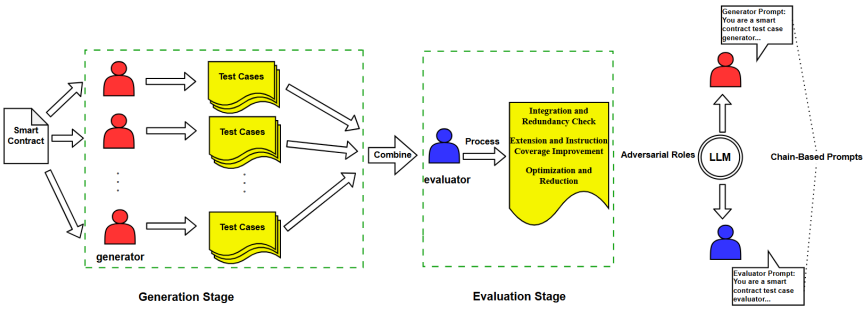

Fig. 4 Adversarial LLM Fuzz Testing Seed Generation

Generation Stage:

In the generation stage, the goal is to generate a sufficient number of initial test cases to cover different code paths and execution scenarios of the smart contract. To achieve this goal, we set up n generator responsible for generating test cases. The output of each generator strictly follows the JSON format to ensure consistency and parseability of the test case structure.

Evaluation Stage:

In the evaluation stage, the generated test cases from the generation stageare merged, optimized, and redundant ones are removed to improve the coverage and quality of the test case set. The objective of this stage is to conduct a thorough review and processing of the generated test cases to ensure their effectiveness and efficiency. The evaluation stage consists of three key steps:

Integration and Redundancy Check (S1):

During actual usage of LLM to generate test cases, we observed that the generated test cases often contain a significant amount of redundancy. If the initial test cases in fuzz testing are too redundant, it can lead to:

1.Inefficient testing: Redundant initial test cases limit the test coverage range as there is significant similarity or repetition among test cases, making it ineffective to explore various aspects of the input space.

2.Inadequate test coverage: Redundant test cases may fail to adequately cover the input space of the target software or system, potentially leaving some vulnerabilities or edge cases undiscovered.

3.Resource waste: Redundant test cases consume additional computational resources and storage space, including CPU time and memory.

4.Misleading results: If the majority of the test cases generated by fuzz testing are redundant, the test results may be misleading, making it difficult to distinguish genuine issues from repetitive ones.

Therefore, in this step, we focus on integrating multiple sets of test data generated by producers and conduct an initial redundancy check. This process aims to optimize the quality of the test cases. Firstly, the test data generated by each producer is integrated. Then, the hash values of each test case are computed using a hash function to generate a unique identifier for each test case. Duplicate test cases are detected and removed by comparing their hash values to improve testing efficiency and resource utilization. The goal of this step is to provide a more refined and diversified test data set for subsequent fuzz testing stages, thereby enhancing testing efficiency and accuracy.

Extension and Instruction Coverage Improvement (S2):

The integrated test cases are passed to LLM, and LLM is requested to extend them. The key to this step lies in leveraging the advantages of adversarial methods to ensure that the generated test cases cover as many code paths and execution scenarios as possible, thereby enhancing the quality and effectiveness of the test case set.

To achieve this goal, a series of adversarial prompts are designed to inspire LLM to generate more challenging test cases. The prompt design is as follows: "You are a evaluator of smart contract test cases, and the above test cases do not achieve 100% coverage. Therefore, I need you to generate more test cases based on the test cases and source code provided by me to achieve 100% instruction coverage for the test case set."

Through such prompts, LLM is guided to focus on generating test cases that can trigger code paths not yet covered. These test cases can help discover potential issues hidden deep in the code and provide more comprehensive test coverage, thereby enhancing the integrity and reliability of the test case set.

Optimization and Reduction (S3):

In the final step, LLM is requested to optimize and reduce the extended test cases to ensure the quality and usability of the test case set are at their best. This step is crucial as it is the last stage of the test case generation process, directly impacting the efficiency and accuracy of subsequent fuzz testing.

In this step, adversarial methods are again employed to guide LLM to perform the following optimization and reduction operations on the test cases:

1.Removal of redundant test cases: LLM is requested to delete test cases with identical coverage paths to reduce the redundancy of the test case set.

2.Modification of non-compliant test cases: LLM is guided to delete and modify test cases that do not comply with specifications, including those with invalid JSON formats and mismatched parameter types and values. The prompt design is as follows: "You are a evaluator of smart contract test cases, and the above test cases contain non-compliant ones. I need you to delete or modify them to comply with the specifications."

Through this process, the aim is to optimize the test case set to contain minimal duplicate test cases and compliant input data, thereby improving the quality of the test cases and the efficiency and accuracy of fuzz testing.

In Section 2.2, a method for generating test cases using chained prompts is proposed: P1 Task Prompt, P2 Context Prompt, P3 Review Prompt, and P4 Optimization Prompt. In the adversarial LLM structure proposed in this section, the execution flow of the producer strictly follows the steps for generating test cases using Chain-Based Prompts as shown in Fig. 3. In the execution flow of the evaluator, since the evaluator's task is to optimize the test cases, this paper decomposes the task prompt P1' of the evaluator into three steps S1, S2, and S3. Additionally, the newly generated test cases by the generator are included as input, as shown in Fig. 5.

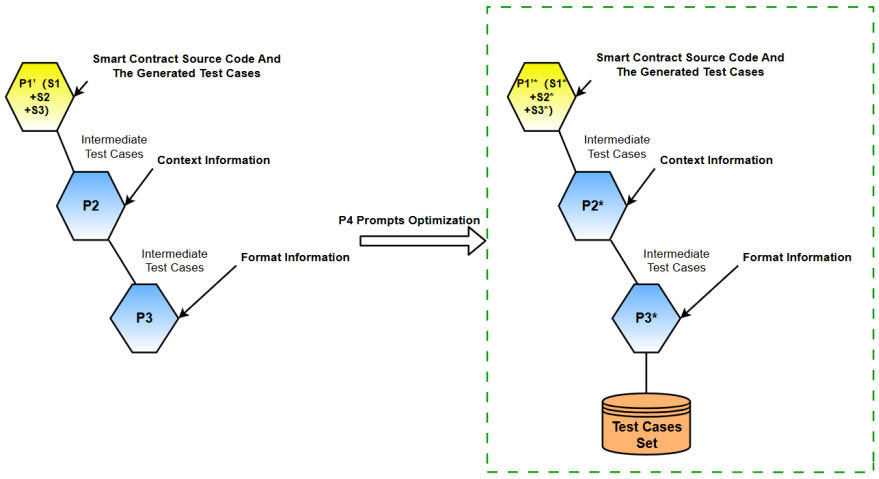

Fig. 5 Adversarial LLM Fuzz Testing Seed Generation

The test cases returned by LLM are in the simplest format, without including information such as contract address, caller address, and function ABI. This is due to the following considerations:

Firstly, these pieces of information can be reused without affecting the experimental results of fuzz testing. In fuzz testing, the contract address, caller address, and function ABI are usually fixed and unchanged during the test. Therefore, they can be omitted during the test case generation process and provided externally, for example, as input parameters to the testing system. This approach has the benefit of reducing the time and computational resources required for generating test cases, thereby improving testing efficiency.

Secondly, focusing LLM on generating more critical information present in actual test cases can enhance the effectiveness of the generated test cases. Key information in fuzz testing typically includes function call parameters, invocation sequence, etc., which directly influence the coverage and quality of test cases. By directing LLM to focus on generating this crucial information, the generated test cases can be more representative and comprehensive, thereby improving the effectiveness of fuzz testing.

It's worth noting that the benchmark fuzz testing system used in this paper is Ityfuzz(Shou et al. 2023), which explores the state space of smart contracts through snapshot states and individual transactions. Ityfuzz proposes a snapshot-based fuzz testing algorithm to reduce the overhead of state re-execution in smart contract fuzz testing. By storing snapshot states and adding snapshots that can increase coverage to the state library, Ityfuzz enables efficient time-traveling in the program and state space search, thereby reducing the time complexity of fuzz testing. Ityfuzz requires a sufficient number of initial test cases at the beginning of testing, which will be used to create snapshots. If the initial test cases fail to trigger critical code paths or cover enough instructions, the generated snapshots may miss some potential states, resulting in insufficient test coverage. Therefore, the coverage of Ityfuzz heavily depends on the quality of initial test cases, which allows for a more intuitive observation of the experimental results in this paper.

The test cases from this system are in standard JSON format data. Therefore, for the test cases generated by LLM, formatting processing is required to meet the seed input requirements of the Ityfuzz system. This includes extracting key information from the test cases and organizing and adjusting them in the seed format of Ityfuzz so that the system can correctly parse and execute these test cases.

5 Experiment

In this section, we will demonstrate the performance improvement brought about by the Adversarial Generation for Smart Contract Fuzz Testing Seeds Guided by Chain-Based LLM. The purpose of this section is to address the following experimental questions:

RQ1: Is the Chain-Based method of optimizing prompts more effective than the original prompts?

In Section 2, this paper introduces a Chain-Based Prompts method for LLM test case generation, which guides LLM to generate test cases through four steps. This design aims to provide LLM with more information to generate more targeted and effective test cases. Therefore, this question mainly explores the extent of improvement in test cases for each step of guidance and verifies it through experiments.

RQ2: Is the inclusion of an adversarial mechanism more efficient than generating test cases without it?

In Section 3, the paper introduces the method of setting up adversarial roles for LLM, aiming to generate higher-quality test cases. This method includes two roles: generator and evaluator, each with n copies. The task of generator is to generate initial test cases based on smart contract source code, while evaluator further improve the generated test cases. Therefore, the key to this question is to explore the extent

of improvement in test case quality introduced by the adversarial mechanism and to determine the optimal number of producers and judges.

RQ3: Does the method proposed in the paper perform better for smart contracts containing various mainstream vulnerabilities?

Atzei et al. reported 12 types of vulnerabilities(Atzei et al. 2017), and some of these vulnerabilities no longer exist as the language iterates. We selected 7 types of vulnerabilities that are still active in the market from these 12 types of vulnerabilities and crawled 30 smart contracts targeting these vulnerabilities on the Ethereum network, forming dataset B3. This question aims to evaluate the effectiveness of the method proposed in the paper in improving instruction coverage when facing mainstream vulnerabilities in smart contracts, in order to determine the potential of the method proposed in the paper to trigger mainstream vulnerabilities in smart contracts.

5.1 Experimental Setup

This study employs Ityfuzz as the benchmark fuzz testing system. Ityfuzz heavily relies on the quality of initial test cases, making it more intuitive to observe the effects of the experiments conducted in this paper. Additionally, GPT-4 is utilized as the LLM to evaluate the performance. As of the time of writing, GPT-4, as the next-generation intelligent model, demonstrates its advantages and robustness. Compared to GPT-3.5, it possesses a larger model scale and parameter count, deeper semantic understanding and reasoning capabilities, more accurate context comprehension and generation, and enhanced self-correction and learning abilities. This enables it to continuously adapt to new language environments and application demands, maintaining a leading position in technology. Considering Ethereum as one of the primary deployment platforms for smart contracts, datasets obtained from this platform exhibit high authenticity. These contracts cover various use cases and functionalities. Moreover, smart contracts on Ethereum are continuously updated, ensuring the timeliness and relevance of the data, reflecting the latest status of the smart contract ecosystem. Therefore, three datasets (B1, B2, and B3) are constructed to evaluate the proposed tool. B1 and B2 consist of 30 and 100 smart contracts respectively, obtained from the Ethereum blockchain. By employing datasets of different scales, the effectiveness of the proposed method under varying contract quantities is validated. The smaller-scale dataset B1 is used for preliminary testing and rapid verification, while the larger-scale dataset B2 better represents the complexity and diversity of the real world, providing more convincing evaluation results. To evaluate RQ3, this paper crawls 30 smart contracts from Ethereum addressing seven types of vulnerabilities that are currently active in the market, forming dataset B3.

5.2 Results Analysis

RQ1: Is the Chain-Based method of optimizing prompts more effective than the original prompts?

To address RQ1, experiments were conducted using dataset B1 as the target dataset, Ityfuzz as the baseline fuzz testing system, and GPT-4 as the LLM model. To verify the hypothesis, five sets of comparative experiments were set up, as shown in

the table 2. Different combinations of prompts were used to request GPT-4 to generate test cases, which were then loaded as initial test cases before fuzz testing. Finally, the performance of the Chain-Based Prompts optimization mechanism was evaluated by comparing the instruction coverage and effective seed proportion after 1 minute of fuzz testing under different prompt combinations. One minute was chosen as the final metric because it was observed during the experiments that the instruction coverage reached its maximum value after 1 minute of fuzz testing.

Table 2 Chain-based Prompt Experiment Results

Combination Of Prompts	Instruction Coverage	Proportion Of Effective Test Cases
P1	31.62%	45.2%
P2+P1	33.17%	47.9%
P2+P1+P3	33.33%	55.4%
P2+P1+P3+P4	34.25%	55.6%
P2+P1+P4	33.94%	48.3%

As shown in Table 2, five sets of comparative experiments were designed in this study to observe the impact of different prompts combinations on the effectiveness of fuzz testing, particularly in enhancing instruction coverage. Initially, the task prompts (P1) generated a series of test cases in fuzz testing, covering some code paths, resulting in an instruction coverage of 31.62%. However, these test cases often lacked in-depth exploration of specific code paths, leading to relatively low coverage.

After introducing context prompts (P2), an increase in instruction coverage to 33.17% was observed. This indicates that context prompts can effectively guide the LLM to better understand and simulate the context of the target system during test case generation, thereby generating more challenging and diverse test cases, and consequently enhancing the coverage of the test case set. Additionally, the proportion of effective test cases increased by 2.7% after adding context prompts, further improving the effectiveness of the test cases.

Following the review of test cases (P3), a significant improvement in the proportion of effective test cases by 7.5% was noted, although the effect on instruction coverage was not significant. This demonstrates that while P3 has a limited direct impact on instruction coverage, its improvements in test case quality still positively influence the effectiveness of fuzz testing.

On the other hand, without adding test case review, the introduction of prompt optimization mechanism (P2 + P1 + P4) resulted in an instruction coverage of 33.94%. This indicates that the introduction of Chain-Based Prompts indeed helps improve the effectiveness of fuzz testing, especially in the initial stages of fuzz testing.

Finally, by adding all prompt combinations (P2 + P1 + P3 + P4), the highest instruction coverage of 34.25% and the highest proportion of effective test cases of 55.6% were observed. These results indicate that the Chain-Based Prompts strategy designed in this study can effectively improve instruction coverage in fuzz testing by guiding the LLM to generate more challenging and diverse test cases, thereby enhancing the quality and effectiveness of the test case set.

RQ2: Is the inclusion of an adversarial mechanism more efficient than generating test cases without it?

To answer RQ2, this experiment evaluates the effectiveness of the adversarial mechanism by posing the following three questions:

1.Can the addition of a generator to fuzz testing improve instruction coverage compared to traditional fuzz testing?

2.Can the simultaneous addition of both generator and evaluatorto fuzz testing improve instruction coverage compared to only adding a generator to fuzz testing?

3.Does increasing the number of generators improve instruction coverage?

Therefore, six sets of comparative experiments were conducted, as shown in Table 3, where A represents the generator, C represents the evaluator, and n represents the number of generators. The experiment involved six participants competing in different environments. Dataset B2 was used in this study, and the relationship between instruction coverage and time during the experiment was collected as the basis for measuring the experimental results. The use of instruction coverage as a basis is because the main goal of the study is to evaluate whether the fuzz testing tool's ability to generate a sufficient number of test cases to cover different statements of the tested program. Instruction coverage is sufficient as an evaluation metric, and previous studies(Shou et al. 2023) have shown a strong correlation between instruction coverage and branch coverage, meaning achieving higher instruction coverage usually leads to higher branch coverage.

Table 3 Adversarial LLM Experiment

Name	Configuration%	Parameter	Description	Instruction Coverage
M1	/	/	Without adding an adversarial mechanism	30.31%
M2	A	n=1	One generator produces a set of test cases	34.25%
M3	A	n=1	Two generators produce two sets of test cases and merge them	35.42%
M4	A+C	n=1	One generator produces a set of test cases; one evaluator optimizes the test cases	35.39%
M5	A+C	n=2	Two generators produce two sets of test cases;one evaluator optimizes the test cases	37.44%
M6	A+C	n=3	Three generators produce three sets of test cases;one evaluator optimizes the test cases	37.28%

As shown in Table 3, when the number of generator set to 2, the instruction coverage reaches its highest value of 37.44%. Additionally, additional experiments were conducted with the methods M1, M3, and M5, which exhibited significant differences in effectiveness, on datasets B1, B2, and B3, respectively, to observe the changes in instruction coverage over time for different methods.

Fig. 6 respectively display the changes in instruction coverage over time for each method on datasets B1, B2, and B3. Here, M1 represents the traditional smart contract fuzz testing method without LLM, M3 represents the smart contract fuzz testing method with two generators, and M5 represents the smart contract fuzz testing method with two generators and one evaluator. It is evident that the method incorporating LLM-generated initial test cases outperforms the traditional method in terms of instruction coverage over time, with a noticeable decrease in the time to reach maximum instruction coverage.

Furthermore, the experiments found that the method utilizing LLM-generated initial test cases could achieve nearly complete instruction coverage within a few seconds when dealing with some simple smart contracts. This rapid coverage capability is a significant advantage of LLM. The analysis in this study suggests that this may be because LLM can more accurately generate initial test cases that cover all statements when dealing with simple smart contracts. This precision enables fuzz testing to explore a wider range of code paths in a short amount of time, thereby accelerating the improvement in instruction coverage. This finding highlights the potential application of LLM in the field of smart contracts, particularly in cases where there is a need to rapidly and effectively identify potential vulnerabilities in code. The initial test cases generated by LLM can significantly enhance the efficiency and accuracy of fuzz testing.

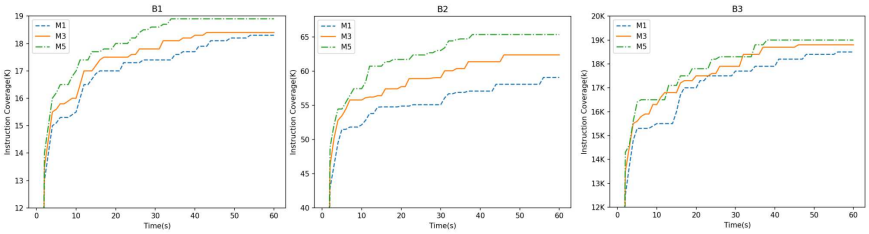

Fig. 6 The variation of instruction coverage over time for different methods

In Section 2.2, the execution process of the "evaluator" was described. To validate the impact of S1, S2, and S3 on the experimental results, the following comparative experiments were designed. Using the B2 dataset, the improvement effect on the experimental results was observed by gradually adding the execution steps of the "evaluator". As shown in Table 4, the effective seed ratio increased from 40.4% to 55.4%, and the instruction coverage at 60 seconds also increased by 2.99%. Furthermore, it can be observed from the table that after adding step S2 (Extension and Instruction Coverage Improvement), there was a significant increase of 1.55% in coverage, demonstrating that step S2 effectively enhances the instruction coverage of the fuzz testing. With the addition of step S3 (Optimization and Reduction), the instruction coverage reached its peak around 30 seconds. Compared to the experiments without step S3, the testing time was shortened by 30 seconds, indicating that step S3 can effectively improve the efficiency of the fuzz testing.

Table 4 Evaluator step-by-step experiment

Step	10s	20s	30s	40s	50s	60s	Proportion Of Valid Seeds
-	21.02%	24.57%	24.57%	24.57%	24.57%	24.57%	24.57%
S1	25.29%	27.11%	28.73%	29.45%	30.37%	31.62%	45.2%
S1+S2	27.53%	29.97%	31.02%	32.76%	33.14%	33.17%	47.9
S1+S2+S3	26.48%	30.36%	33.33%	33.33%	33.33%	33.33%	55.4

GPT-4 has a temperature parameter in its invocation parameters, which takes a floating-point value between 0 and 1, with a default of 0.7. When the temperature value is close to 0, the model generates more deterministic and consistent text, potentially leading to stronger repetition, meaning it tends to choose the highest probability output. This may result in content that is overly fixed and uniform. Conversely, when the temperature value approaches 1, the model generates more diverse and unordered text, considering more possibilities when selecting the next word or phrase. This can make the generated content more varied and creative, but it may also lead to unpredictable or incoherent model outputs.

To assess the impact of temperature on the generated test cases and the coverage of fuzz testing, this study conducted the following experiments: For each target program, the approach was continuously run on the B2 dataset for 2 hours. At each stage, the sampling temperature was set to values between 0 and 1 with a step size of 0.1. At the end of each experiment, three metrics were used to evaluate the quality of the generated seeds: 1) uniqueness ratio of test cases, 2) effectiveness ratio of test cases, and 3) code coverage rate.

Test Case Uniqueness Ratio:

As shown in Fig. 9, the proportion of unique test cases increases with the increase in temperature. When the sampling temperature is below 0.2, close to 90% of the test cases are repetitive. Generally, as the sampling temperature increases, more unique test cases can be generated. Across the three datasets, the test case uniqueness ratio increases with the sampling temperature. When the temperature exceeds 0.7, it was observed that within this temperature range, more test cases with format errors or parameters completely inconsistent with requirements are generated.

Test Case Validity Ratio:

The paper also evaluated the ratio of syntactically valid seeds generated under different sampling temperature settings for each configuration. The criterion for determining the validity of test cases is whether they can be loaded into the Ityfuzz model after conversion. According to the results in Fig. 7, the proportion of valid test cases decreases as the sampling temperature increases. The reason is that with the increase in sampling temperature, the model's output becomes more random, and therefore, it is more likely to violate the format rules of test cases.

Improvement in Coverage:

For each model, different sampling temperatures ranging from 0 to 1 were evaluated, with a step size of 0.1. For each configuration, the instruction coverage corresponding to nine different temperature values was evaluated. At the end of the

fuzz testing experiment, the instruction coverage for each temperature was calculated. The results, as shown in Fig. 8, indicate that the code coverage is highest at a temperature of 0.7.

Through the analysis of the three sets of experiments, it was found that at lower temperatures, the proportion of valid test cases is highest, but the model generates a large number of repetitions, resulting in lower improvement in code instruction coverage. On the other hand, at higher temperatures, the proportion of unique test cases is highest, but the model generates more syntactically invalid test cases, which contradicts the purpose of using LLM to generate syntactically valid test cases. Therefore, a temperature of 0.7 is identified as the optimal temperature value for the method proposed in this paper, demonstrating that for this method, 0.7 is the optimal temperature value.

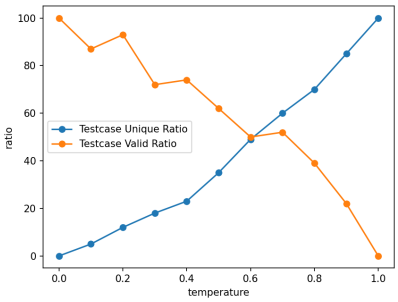

Fig. 7 The variation of test cases with temperature

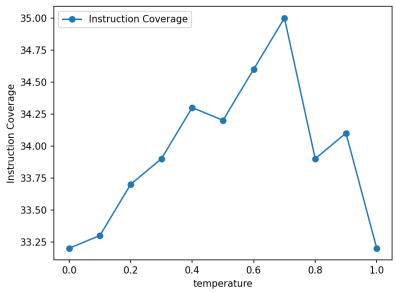

Fig. 8 Instructions coverage variation with temperature

RQ3: Does the method proposed in the paper perform better for smart contracts containing various mainstream vulnerabilities?

To address RQ3, this paper conducted comparative experiments on smart contracts in the B3 dataset containing various types of vulnerabilities, as shown in Table 5. M1 represents fuzz testing without any configuration, while M5 represents the experiment configuration with the best performance in answering RQ2. The method proposed in this paper shows a 1.2% improvement in average instruction coverage. However, this

improvement varies across different types of vulnerabilities. Specifically, after conducting separate analyses for each vulnerability type, it was observed that the effectiveness of LLM-generated test cases is influenced by the complexity of vulnerabilities. The improvement in instruction coverage achieved by the method proposed in this paper decreases as the complexity of vulnerabilities increases.

Table 5 Improvement in Coverage Rates for Various Types of Vulnerabilities

Vulnerabilities Name	M1	M5
Gasless Send	35.6%	36.2%
Exception Disorder	22.6%	22.8%
Reentrancy	12.7%	12.9%
Timestamp Dependency	28.3%	31.6%
Block Number Dependency	25.1%	28.9%
Dangerous Delegatecall	15.6%	16.0%
Freezing Ether	24.2%	24.7%
Average	23.4%	24.7%

For each of the seven different vulnerability types mentioned above, here is a specific analysis:

(1) Gasless Send Vulnerability: This vulnerability occurs when executing transfer operations in smart contracts without correctly setting gas limits, allowing attackers to perform gasless transfers. In the experiment, the mechanism of incorporating LLM for generating test cases showed a 0.6% improvement in instruction coverage rate.

(2) Exception Disorder Vulnerability: Exception disorder refers to security issues caused by incorrect handling of exception sequences in smart contracts. This vulnerability may arise from logic errors in exception handling mechanisms, allowing attackers to bypass exception handling and execute malicious operations. In the experiment, the effectiveness of LLM-generated test cases for improving exception disorder vulnerabilities was not significant, likely due to the need for deeper code path exploration and complex exception handling scenarios.

(3) Reentrancy Vulnerability: Reentrancy is one of the most common security issues in smart contracts, typically occurring when state updates and external calls are not properly handled. In the experiment, the mechanism of incorporating LLM for generating test cases did not significantly improve the instruction coverage rate, as reentrancy vulnerabilities often require specific contextual environments and complex operation sequences to trigger, which LLM may not accurately simulate.

(4) Timestamp Dependency Vulnerability: This vulnerability involves security issues in smart contracts that rely on blockchain timestamps, allowing attackers to manipulate timestamps to execute attacks. In the experiment, LLM-generated test cases showed a 3.3% increase in instruction coverage rate. Furthermore, LLM-generated test cases directly covered the code paths triggering the vulnerabilities, significantly reducing fuzz testing time.

(5) Block Number Dependency Vulnerability: This vulnerability involves security issues in smart contracts that rely on block numbers, allowing attackers to manipulate block numbers to execute attacks. In the experiment, LLM-generated test cases showed a 3.8% increase in instruction coverage rate. Similar to timestamp dependency vulnerabilities, LLM-generated test cases directly covered the code paths triggering the vulnerabilities.

(6) Dangerous Delegatecall Vulnerability: This vulnerability arises from the use of insecure delegatecall instructions in smart contracts, allowing attackers to execute malicious operations. In the experiment, the improvement in LLM-generated test cases for dangerous delegatecall vulnerabilities was not significant because LLM cannot accurately simulate the context generated by delegate calls, failing to capture the accessed and modified state of the current contract by the called contract.

(7) Freezing Ether Vulnerability: This vulnerability results from improper handling of transfer operations in smart contracts, allowing attackers to freeze ether in contracts. While LLM can generate some specific test cases to simulate special transactions or invocation paths that attackers may exploit, fully covering freezing ether vulnerabilities may require more complex and specific test cases, which LLM may not always generate. Therefore, despite LLM's ability to improve instruction coverage to some extent, there may still be cases where vulnerabilities are not fully revealed.

These findings reveal that the effectiveness of LLM-generated test cases depends on the complexity of vulnerabilities. For relatively simple and easily triggered vulnerabilities, LLM's generation strategy can quickly and effectively produce test cases with high coverage rates, even directly generating test cases triggering vulnerabilities. However, for more complex vulnerabilities or those requiring specific contextual environments, LLM's effectiveness may be limited because it cannot accurately simulate the specific conditions or operation sequences required to trigger vulnerabilities. This issue is expected to be addressed with the enhancement of LLM's computing power and continuous improvement and optimization of LLM's seed generation algorithm, leading to further improvements in LLM's performance in generating test cases for more complex vulnerabilities.

6 Conclusion

This paper proposes a method for utilizing LLM to generate initial seed sets for smart contract fuzz testing, aiming to enhance the coverage of traditional fuzz testing. Through experiments on 160 real-world smart contracts, it was found that the proposed method can effectively improve the efficiency and instruction coverage of fuzz testing, providing new ideas and methods for the evaluation of smart contract security.

As for future work, we plan to further enhance the coverage of fuzz testing under complex smart contracts. One possible approach is to instrument the smart contract code to submit specific coverage information to LLM after each fuzz testing execution ends, assisting in analysis. This operation can help LLM understand the coverage of seeds more accurately, thereby generating more effective seeds. Additionally, we also plan to expand the code of this paper to adapt to more fuzz testing tools, further enhancing the comprehensiveness and effectiveness of testing.

Acknowledgements. This work was supported in part by the Key R & D project of Shaanxi Province(2023-YBGY-030), the Key Industrial Chain Core Technology Research Project of Xi'an (23ZDCYJSGG0028-2022), the National Natural Science Foundation of China (62272387).

References

[1] Atzei, N., Bartoletti, M., Cimoli, T.: A Survey of Attacks on Ethereum Smart Contracts (SoK). In: Maffei, M., Ryan, M. (eds) Principles of Security and Trust. POST 2017. Lect. Notes Comput. Sci. 10204. Springer, Berlin, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54455-6_8

[2] Jiang, B., Liu, Y., Chan, W.K.: ContractFuzzer: Fuzzing Smart Contracts for Vulnerability Detection. 2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE), Montpellier, France (2018). https://doi.org/10.1145/3238147.3238177

[3] Liu, C., Liu, H., Cao, Z., Chen, Z., Chen, B., Roscoe, B.: ReGuard: Finding Reentrancy Bugs in Smart Contracts. IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion), Gothenburg, Sweden, 2018, pp. 65-68 (2018).

[4] Wustholz, V., Christakis, M.: Harvey: A greybox fuzzer for smart contracts.Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. 2020, pp. 1398-1409.(2020)

[5] Shou, C., Tan, S., Sen, K.: Ityfuzz: Snapshot-based fuzzer for smart contract.Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis, 2023, pp. 322-333.(2023)

[6] Sakhrawi Z, Labidi T. Test case selection and prioritization approach for automated regression testing using ontology and COSMIC measurement[J]. Automated Software Engineering, 2024, 31(2): 51.(2024)

[7] Miah, M.S.U., Kabir, M.M., Sarwar, T.B. et al. A multimodal approach to cross-lingual sentiment analysis with ensemble of transformer and LLM. Sci Rep 14, 9603 (2024).https://doi.org/10.1038/s41598-024-60210-7

[8] Wu T, He S, Liu J, et al. A brief overview of ChatGPT: The history, status quo and potential future development[J]. IEEE/CAA Journal of Automatica Sinica, 2023, 10(5): 1122-1136.(2023)

[9] Feng Z, Guo D, Tang D, et al. Codebert: A pre-trained model for programming and natural languages.(2020) arXiv preprint arXiv:2002.08155

[10] Guo D, Ren S, Lu S, et al. Graphcodebert: Pre-training code representations with data flow.(2020) arXiv preprint arXiv:2009.08366

[11] Wang Y, Wang W, Joty S, et al. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation.(2021) arXiv preprint arXiv:2109.00859

[12] Enis M, Hopkins M. From LLM to NMT: Advancing Low-Resource Machine Translation with Claude.(2024) arXiv preprint arXiv:2404.13813

[13] Mihalache A, Grad J, Patil N S, et al. Google Gemini and Bard artificial intelligence chatbot performance in ophthalmology knowledge assessment. Eye, 2024: 1-6.(2024)

[14] Zolkepli H, Razak A, Adha K, et al. Large Malaysian Language Model Based on Mistral for Enhanced Local Language Understanding.(2024) arXiv preprint arXiv:2401.13565

[15] Radhakrishnan A M. Is Midjourney-AI a new anti-hero of architectural imagery and creativity. GSJ, 2023, 11(1): 94-104.(2023)

[16] Borji A. Generated faces in the wild: Quantitative comparison of stable diffusion, midjourney and dall-e 2.(2022) arXiv preprint arXiv:2210.00586

[17] Xia, Chunqiu Steven, et al. Fuzz4all: Universal fuzzing with large language models[C]//Proceedings of the IEEE/ACM 46th International Conference on Software Engineering. 2024: 1-13.(2024)

[18] Hu, Jie, Qian Zhang, and Heng Yin. Augmenting greybox fuzzing with generative ai.(2023) arXiv preprint arXiv:2306.06782

[19] Ackerman J, Cybenko G. Large language models for fuzzing parsers (registered report)[C]//Proceedings of the 2nd International Fuzzing Workshop. 2023: 31-38.(2023)

[20] Deng Y, Xia C S, Peng H, et al. Large language models are zero-shot fuzzers: Fuzzing deep-learning libraries via large language models[C]//Proceedings of the 32nd ACM SIGSOFT international symposium on software testing and analysis. 2023: 423-435.(2023)

[21] Deng Y, Xia C S, Yang C, et al. Large language models are edge-case fuzzers: Testing deep learning libraries via fuzzgpt.(2023) arXiv preprint arXiv:2304.02014

[22] Zhang C, Bai M, Zheng Y, et al. Understanding large language model based fuzz driver generation.(2023) arXiv preprint arXiv:2307.12469

[23] Lemieux C, Inala J P, Lahiri S K, et al. Codamosa: Escaping coverage plateaus in test generation with pre-trained large language models[C]//2023 IEEE/ACM

45th International Conference on Software Engineering (ICSE). IEEE, 2023: 919-931.(2023)

[24] Gu Q. Llm-based code generation method for golang compiler testing[C]//Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering. 2023: 2201-2203.(2023)

[25] Qiu F, Ji P, Hua B, et al. Chemfuzz: Large language models-assisted fuzzing for quantum chemistry software bug detection[C]//2023 IEEE 23rd International Conference on Software Quality, Reliability, and Security Companion (QRS-C). IEEE, 2023: 103-112.(2023)

[26] Ma W, Wu D, Sun Y, et al. Combining Fine-Tuning and LLM-based Agents for Intuitive Smart Contract Auditing with Justifications.(2024) arXiv preprint arXiv:2403.16073

[27] Meng R, Mirchev M, Bohme M, et al. Large language model guided protocol fuzzing[C]//Proceedings of the 31st Annual Network and Distributed System Security Symposium (NDSS). 2024.(2024)

[28] Wei J, Tay Y, Bommasani R, et al. Emergent abilities of large language models.(2022) arXiv preprint arXiv:2206.07682

[29] Hu, Sihao, et al. Large language model-powered smart contract vulnerability detection: New perspectives.(2023) arXiv preprint arXiv:2310.01152

[30] Liu C, Bao X, Zhang H, et al. Improving chatgpt prompt for code generation.(2023) arXiv preprint arXiv:2305.08360

[31] Chowdhery A, Narang S, Devlin J, et al. Palm: Scaling language modeling with pathways[J]. Journal of Machine Learning Research, 2023, 24(240): 1-113.(2023)

[32] Goodfellow I, Pouget-Abadie J, Mirza M, et al. Generative adversarial nets[J]. Advances in neural information processing systems, 2014, 27.(2014)

[33] Atzei N, Bartoletti M, Cimoli T. A survey of attacks on ethereum smart contracts (sok)[C]//Principles of Security and Trust: 6th International Conference, POST 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings 6. Springer Berlin Heidelberg, 2017: 164-186.(2017)