Poster: *FuzzLGen*: Logical Seed Generation for Smart Contract Fuzzing via LLM-based Agents and Program Analysis

Songyan Ji, Mingyue Xu, Jin Wu, Jian Dong
Harbin Institute of Technology
songyan96@hit.edu.cn, mingyue81@stu.hit.edu.cn, wujin@hit.edu.cn, dan@hit.edu.cn

***Abstract*—Smart contracts play a pivotal role in blockchain applications but are increasingly targeted by attackers, resulting in significant financial losses. As their adoption grows across various industries, ensuring their security has become more important than ever. Fuzzing has emerged as an effective technique for detecting vulnerabilities in smart contracts. However, a major limitation of current fuzzing methods is their struggle to generate a high-quality initial seed corpus, leading to low code coverage, which is crucial for comprehensive vulnerability detection. To address this, we propose *FuzzLGen*, a novel framework that integrates LLM-based agents with program analysis to generate an initial seed corpus. This approach not only enhances fuzzing effectiveness but also resolves the reliability issues associated with LLM-based vulnerability detection by leveraging fuzz execution. We evaluated *FuzzLGen* by applying it to a state-of-the-art smart contract fuzzer. The experimental results demonstrate that the high-quality initial seed corpus generated by *FuzzLGen* significantly boosts the efficiency of fuzzing, highlighting the framework's effectiveness in enhancing smart contract security.**

I. Motivation

A critical factor influencing fuzzing effectiveness is constructing a high-quality initial seed [1]. A well-crafted seed can produce valuable mutations, enabling deeper path exploration and helping to uncover vulnerabilities. However, recent experimental reviews [2] indicate that current fuzzing techniques for smart contracts struggle with generating effective initial seed corpus, limiting their ability to achieve comprehensive path coverage. Given this, it is crucial to optimize the generation of high-quality initial seed corpus for smart contracts.

Recent developments in Large Language Models (LLMs) have introduced new insights and potential solutions in software engineering, which may lead to innovative approaches for generating initial seeds in smart contract fuzzing. However, directly querying LLMs for seed generation tends to produce high false positives and negatives, due to the intricate dependencies of smart contract variables, LLM hallucinations, and the infeasibility of feeding multiple Solidity code files into LLMs. To solve these problems, we propose *FuzzLGen*, the first seed generation framework for smart contract fuzzing that combines LLM-based agents and program analysis.

II. Challenges and Approach

We encountered three key challenges during the design and implementation of *FuzzLGen*: **C1:** Smart contract projects often consist of multiple Solidity files, making it impractical or costly to process all of them through LLMs directly. While prior work has used specific function filtering techniques to reduce contract size for LLM applications [3], this approach is inadequate for contract size reduction in coverage analysis. Removing certain functions may result in the loss of essential dependencies required for properly executing various branches within the contract. **C2:** Directly using LLMs struggles to accurately generate transaction sequences that satisfy the intricate branch constraints in contracts, which are difficult to cover due to the complex and potential circular dependencies between state variables. **C3:** LLMs may encounter hallucination issues, generating results that do not align with the transaction specifications, making them unusable by the fuzzer.

To address these challenges, we design *FuzzLGen*, a solution that combines multiple LLM-based agents with program analysis. The input to *FuzzLGen* is a smart contract project, and its output is an initial seed corpus for the fuzzer. The architecture of *FuzzLGen*, shown in Figure 1, is explained in detail below. To address Challenge 1, we first eliminate code that poses no security risks, such as verified library code, and remove non-essential comments. For cases where the contract remains too large, we employ static dependency analysis to group functions based on their interdependencies. This approach identifies and captures branch dependencies, where a variable is read in one function and written in another, as well as function call dependencies, which are represented as a directed graph. After transforming this graph into an undirected one, we can cluster related functions, preserving the contract's logic and branch coverage while reducing its size. To address Challenge 2, the key idea is to enhance the LLM's understanding of Solidity code, especially in handling circular dependencies between state variables. To achieve this, we first identify the basic control flow structure within the Solidity code, which can then be utilized to generate valid transaction sequences. We design the **BlockAnalyzer** agent for this task. BlockAnalyzer identifies basic blocks that do not contain jump instructions and is equipped with knowledge of Solidity syntax. It recognizes critical constructs like `require`, `assert`, and function calls, which can alter the execution flow. The agent outputs the basic blocks, providing essential information that enables the PathDesigner agent to generate valid transaction sequences. The **PathDesigner** agent is tasked with generating transaction sequences, guided by restrictions

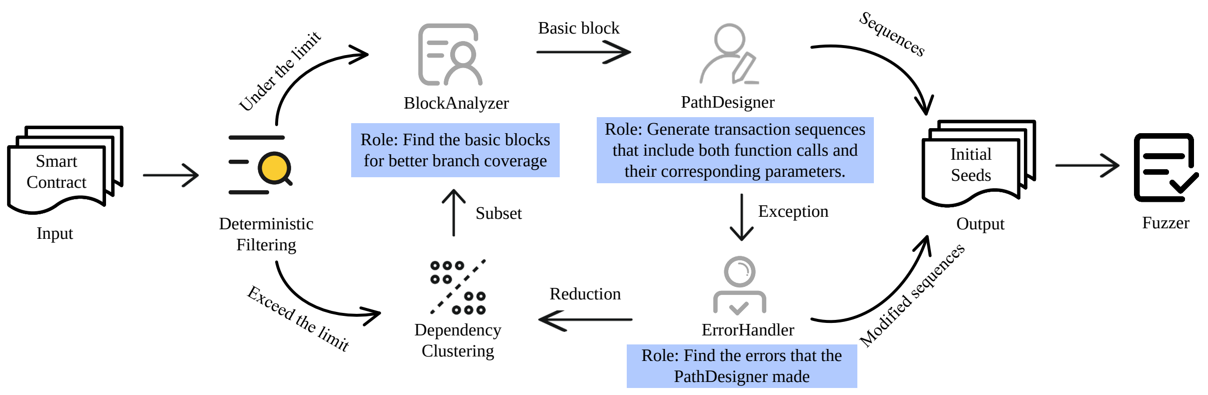


Fig. 1. Overview of *FuzzLGen*.

we define, such as condition validations, function call dependencies, and proper variable initializations. These constraints ensure the validity of transaction paths. The output consists of transaction sequences, including function parameters and transaction arguments. To address Challenge 3, we create the **ErrorHandler** agent, designed to identify and resolve errors in the sequences generated by PathDesigner. These errors can stem from hallucination issues, which prevent the fuzzer from executing the sequences, or from results exceeding the allowed output length. The ErrorHandler agent resolves these issues by taking one of two actions: Reduction or Modify. The Reduction action shortens the result when its length exceeds the prompt's output limit. The Modify action involves correcting the result and then regenerating the seed.

III. Evaluation

To validate the effectiveness of the initial seeds generated by *FuzzLGen*, we utilized them as input for Smartian [4], a state-of-the-art fuzzer, which achieves higher code coverage than other tools [2]. The experiment compared the vulnerability detection performance under three conditions: using Smartian's default seed corpus, the *FuzzLGen*-generated seed corpus (denoted as *LSeedSmartian*), and the seed corpus directly generated by LLMs (denoted as *LLMSmartian*). All setups adhered to Smartian's vulnerability detection framework, with consistent execution time (one hour per contract), a benchmark dataset of 500 contracts, and vulnerability definitions, ensuring a fair comparison. Figure 2 presents the results. The upper figure shows the variation in the number of vulnerabilities detected by the fuzzer over time, depending on the initial seeds used. Notably, within the same timeframe, *FuzzLGen* detects more vulnerabilities, with a 22.4% improvement over Smartian. In contrast, the LLM-generated seeds show only a small performance improvement over Smartian, with an increase of just 4.7%. While LLM introduces new insights and potential solutions for software engineering, its direct application yields limited improvement. The lower figure shows the ability to detect vulnerabilities across different vulnerability categories. *FuzzLGen* outperforms Smartian in most of these categories.

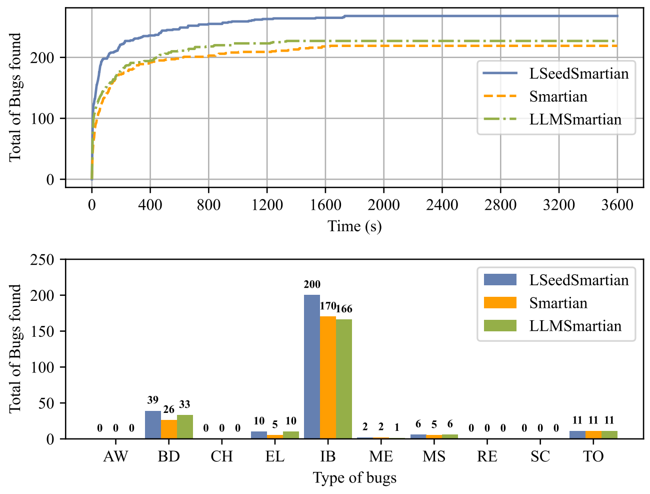


Fig. 2. Number of bugs found by different initial seed corpus.

IV. Conclusion

In this paper, we proposed *FuzzLGen*, the first seed generation framework for smart contract fuzzing via LLM-based agents and program analysis. We applied it to a state-of-the-art smart contract fuzzer, and the experimental results demonstrate that a high-quality initial seed corpus generated by *FuzzLGen* substantially enhances the effectiveness of the fuzzer.

References

[1] A. Herrera, H. Gunadi, S. Magrath, M. Norrish, M. Payer, and A. L. Hosking, "Seed selection for successful fuzzing," in *Proceedings of the 30th ACM SIGSOFT ISSTA*, 2021, p. 230–243.

[2] S. Wu, Z. Li, L. Yan, W. Chen, M. Jiang, C. Wang, X. Luo, and H. Zhou, "Are we there yet? unraveling the state-of-the-art smart contract fuzzers," in *Proceedings of the IEEE/ACM 46th ICSE*, 2024.

[3] Y. Sun, D. Wu, Y. Xue, H. Liu, H. Wang, Z. Xu, X. Xie, and Y. Liu, "Gptscan: Detecting logic vulnerabilities in smart contracts by combining gpt with program analysis," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 2024, pp. 1–13.

[4] J. Choi, D. Kim, S. Kim, G. Grieco, A. Groce, and S. K. Cha, "Smartian: Enhancing smart contract fuzzing with static and dynamic data-flow analyses," in *2021 36th IEEE/ACM International Conference on ASE*, pp. 227–239.

Poster: *FuzzLGen*: Logical Seed Generation for Smart Contract Fuzzing via LLM-based Agents and Program Analysis

Songyan Ji, Mingyue Xu, Jin Wu, Jian Dong
Harbin Institute of Technology
songyan96@hit.edu.cn, mingyue81@stu.hit.edu.cn, wujin@hit.edu.cn, dan@hit.edu.cn

Introduction

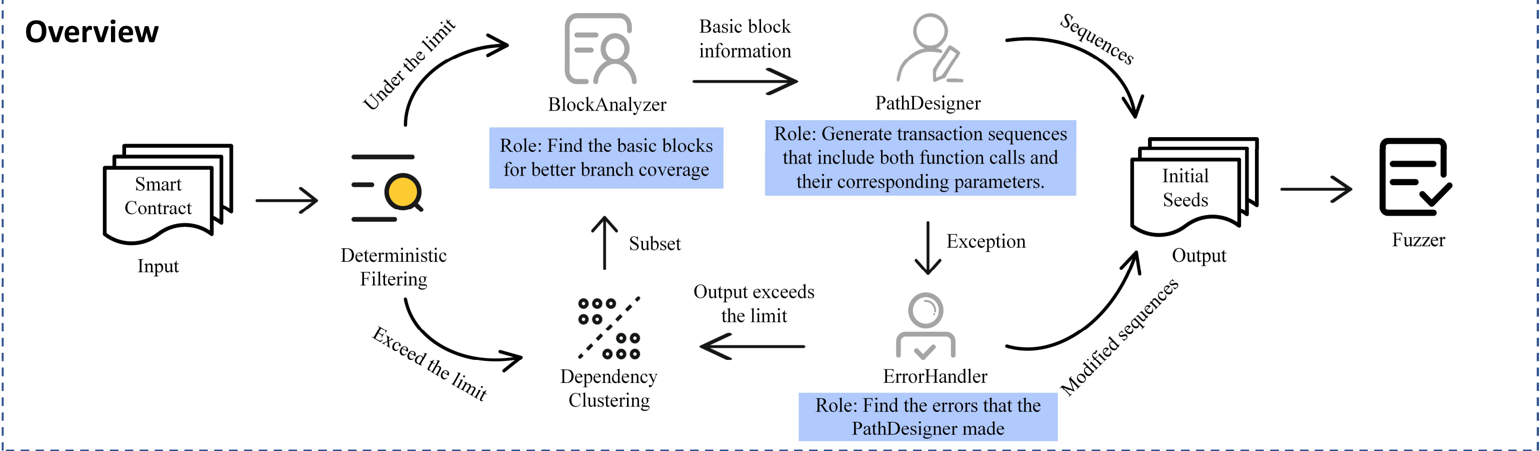
- Current smart contracts fuzzing techniques struggle with generating effective initial seed corpus [1].
- Large Language Models (LLMs) have introduced new insights and potential solutions in software engineering, yet their application to smart contract seed generation remains unexplored and challenging.
- We proposed *FuzzLGen*, the first seed generation framework for smart contract fuzzing that combines LLM-based agents and program analysis.

Challenges

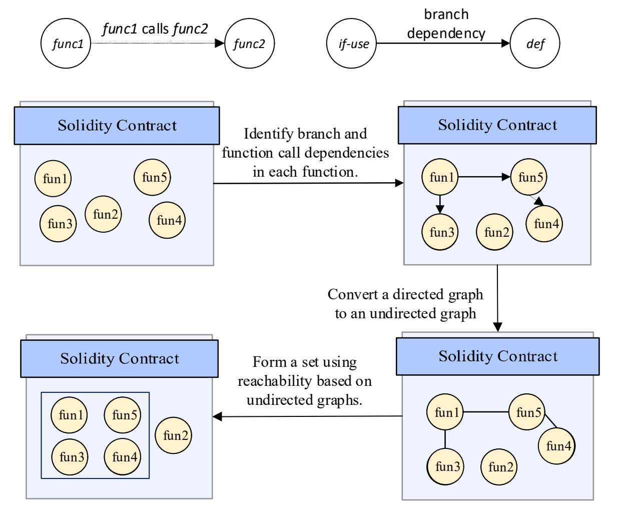
- C1: Smart contracts often consist of multiple Solidity files, making direct LLM processing impractical. While existing function-filtering methods can reduce contract size [2], they do not effectively support LLM-based coverage analysis.
- C2: LLMs struggle to generate transaction sequences that accurately satisfy complex branch constraints, hindered by intricate and circular dependencies among state variables.
- C3: LLMs may produce outputs misaligned with transaction specifications, rendering them unusable for fuzzing.

FuzzLGen Design

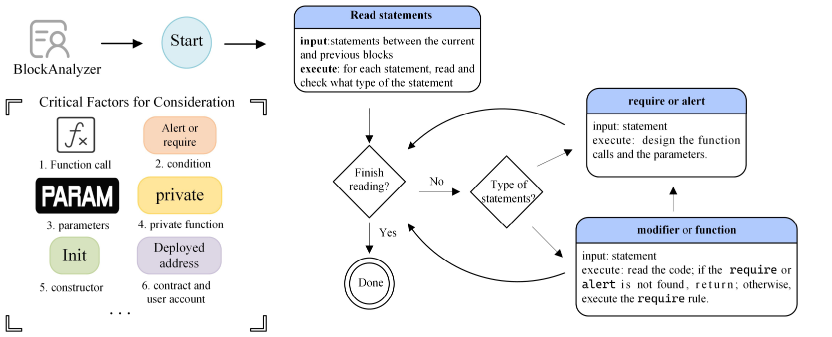
Overview



Dependency Clustering

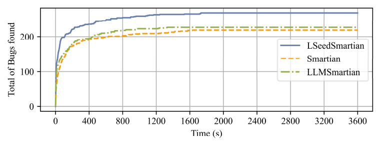

func1
func1 calls func2
func2
if-use
branch dependency
def
Solidity Contract
fun1
fun2
fun3
fun4
fun5
Identify branch and function call dependencies in each function.
Convert a directed graph to an undirected graph
Form a set using reachability based on undirected graphs.

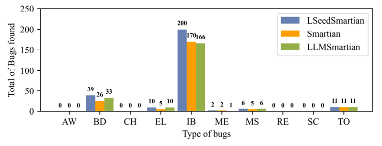
Generate transaction sequences


BlockAnalyzer
Start
Read statements
input:statements between the current and previous blocks
execute: for each statement, read and check what type of the statement
Critical Factors for Consideration
1. Function call
Alert or require
2. condition
PARAM
3. parameters
private
4. private function
Init
5. constructor
Deployed address
6. contract and user account
. . .
Finish reading?
No
Yes
Done
Type of statements?
require or alert
input: statement
execute: design the function calls and the parameters.
modifier or function
input: statement
execute: read the code; if the require or alert is not found, return; otherwise, execute the require rule.

Evaluation

- FuzzLGen detects more vulnerabilities, with a 22.4% increase over Smartian and an 18.1% increase over the direct use of LLM.
- In the same timeframe, FuzzLGen detected more vulnerabilities.
- FuzzLGen outperforms Smartian in most vulnerability categories.





Reference
[1] S. Wu et al., “Are we there yet? Unraveling the state-of-the-art smart contract fuzzers,” in Proceedings of the IEEE/ACM 46th ICSE, 2024.
[2] Y. Sun et al., “Gptscan: Detecting logic vulnerabilities in smart contracts by combining GPT with program analysis,” in Proceedings of the IEEE/ACM 46th International Conference on Software Engineering, 2024, pp. 1–13.