Improving Smart Contract Code with LLMs

Radosław Klimek
AGH University of Krakow
Krakow, Poland *rklimek@agh.edu.pl*

Abstract

Smart contracts are pivotal in blockchain systems, yet ensuring their reliability and security remains challenging due to coding complexities and potential vulnerabilities. This paper explores the use of Large Language Models (LLMs) in enhancing the smart contract code quality. As part of leveraging extensive training data and language understanding, we experiment with different approaches. LLMs aid developers by offering automated code suggestions, identifying vulnerabilities and promoting best practices. Through experimentation, we demonstrate how integrating LLM-based approaches improves code quality and reliability in blockchain applications.

Keywords: Smart contract, Large Language Model, verification.

1. Introduction

Smart contracts streamline transactions by automating them, eliminating the need for intermediaries and reducing costs. By leveraging blockchain technology, they ensure immutability and indisputability, thus enhancing transaction security. Their application accelerates processes, thereby enhancing economic efficiency. Detecting errors in smart contracts is crucial due to the practical irreversibility of deployed contracts. Even small errors can lead to permanent financial losses or other serious consequences. Therefore, it is important to ensure that smart contracts are error-free and operate as expected.

LLMs (Large Language Models) are remarkable and novel tools with a wide range of applications. In addition to generating high-quality texts and aiding decision-making processes, LLMs can also be used for verifying smart contracts, offering an innovative approach to enhancing security in blockchain-based financial applications. The aim of this article is to review methods and tools for verifying smart contracts and experiments aimed at using LLMs for contract verification.

The approach presented in this study is innovative and has garnered limited attention in the existing literature [1]. Zhao et al. [4] discuss the generation of comments for smart contracts using a language model, emphasising the selection of the most appropriate contract fragments from a historical dataset. Similarly, Shou et al. [3] propose an approach that utilizes specific metrics to adjust the number of generated test cases based on the contract's complexity. Ma et al. [2] introduce a tool that facilitates both the tuning and auditing of contracts.

2. Preparations

There are several methods and strategies for verifying smart contracts, including:

- Static Analysis – involves reviewing the contract code to identify potential errors and inconsistencies before deployment. This is often done using static code analysis tools;
- Dynamic Analysis – involves conducting tests that simulate various usage scenarios of the contract to check if it behaves as expected. These tests can be both manual and automated; and
- others, like formal verification (mathematical techniques), community verification (pub-

lishing the contract code in open repositories), etc.

We will focus on and limit ourselves to the first two approaches.

Static analysis is the process of examining source codes to detect errors or irregularities. This analysis is performed without actually executing the program, meaning the analyser does not need access to the program's runtime environment. Instead, the code is thoroughly reviewed and analysed using computer-based tools. It can detect errors not visible during program execution, such as syntactic, semantic, and logical errors that may cause unexpected program behaviuor. The static code analysis can be applied to analyse an entire program, which may be difficult or impossible during dynamic testing. We tested the following frameworks: Securify, Solhint, Slither, Mythril, Manticore, Echidna and Oyente. *Dynamic analysis* is an approach to testing smart contracts in the real blockchain execution environment with behaviuor monitoring. It can detect errors not visible during a static analysis and can be used to test interactions between the different parts of codes. It can analyse contract performance by identifying whether it exceeds the gas limit in a block, which can result in its blocking. In all experiments, we analysed contracts written in the Solidity language. Faulty contracts were sourced randomly from the publicly available repositories. We considered twenty contracts.

3. Experiments and evaluation

To analyse and possibly repair smart contracts, the following language models were utilised: Llama, Cohere and TextCortext. For each smart contract from the considered database of smart contracts, individual language models were requested to repair the contract by providing the following data:

- Smart contract code,
- Test result from each testing tool (e.g., Slither),
- Command to return the entire corrected smart contract (or fragment).

Experiments where the entire contract was provided as input were deemed less satisfactory. The quality of responses was lower, and the returned contracts often contained compilation errors. Therefore, we changed our approach and provided contract fragments, individual instructions, entire conditional statements, or loops as input data, see Figure 1. LLM models performed better with this approach, i.e., with contract fragments[1]. We obtained a series of suggestions that we consider valuable and which enable us to improve the contract code.

The analysis of the investigated tools for smart contract verification revealed significant diversity in terms of functionality and effectiveness in detecting errors and vulnerabilities. Approximately twenty contracts available in various repositories were tested. An intriguing aspect of the experiments was the application of language models such as Llama, Cohere, and TextCortex to improve smart contracts in terms of compilation errors and test results. It was found that the use of these models led to a significant reduction in the number of errors. Additionally, the language models provided a range of suggestions and were able to autonomously fix contracts with very high effectiveness. The average correctness rates were as follows: 74% for the Llama model, 75% for Cohere and 79% for TextCortex, indicating the value of these models in the process of smart contract testing. They enable the minimisation of error risks and the enhancement of contract quality. In summary, comments obtained from LLMs are valuable and promising. They encourage further research into the development of supporting tools. Such interaction of

[1] I would like to express my gratitude to the following students Seweryn Dutka, Arkadiusz Kupiec, Justyna Kurcz and Jakub Piekielniak for their support in the experimental works.

Error	Frequency	Cohere	Cohere	Llama	Llama	Cortex	Cortex
IncorrectSolidityVersion	19	16	84%	13	68%	17	89%
AssemblyUsage	1	0	0%	1	100%	1	100%
NonMixedCaseFunctionName	4	3	75%	3	75%	3	75%
Rule is set with explicit type [var/s: uint]	11	10	91%	8	73%	5	45%
A public function that is never called within the contract should be marked as external	9	7	78%	6	67%	8	89%
NoVisibilitySpecified	5	5	100%	3	60%	4	80%
DefiningConstructorsAsFunctions	5	1	20%	4	80%	4	80%
Provide an error message for revert	1	0	0%	1	100%	1	100%
Provide an error message for require	7	3	43%	5	71%	5	71%
Use Custom Errors instead of revert statements	2	1	50%	1	50%	1	50%
Explicitly mark visibility of state	6	3	50%	4	67%	5	83%
Explicitly mark visibility in function	6	4	67%	4	67%	5	83%
Defining constructors as functions with the same name as the contract is deprecated. Use "constructor(...) { ... }" instead.	3	1	33%	2	67%	2	67%
Line length must be no more than 80 but current length is 81	1	0	0%	1	100%	1	100%
NonMixedCaseParameterName	4	3	75%	0	0%	2	50%
Use Custom Errors instead of require statements	9	4	44%	5	56%	8	89%
StateVariableShouldBeImmutable	2	1	50%	1	50%	1	50%
Parameter EchidnaTestTimeAndCaller.setSender(address)._sender is not in mixedCase	1	0	0%	0	0%	1	100%
State variables that do not change should be declared as constants.	1	1	100%	0	0%	1	100%
A variable is declared but never initialized.	1	1	100%	0	0%	1	100%
Contract fields that can be modified by any user must be inspected.	5	1	20%	3	60%	4	80%
Returned value relies on block timestamp.	2	0	0%	2	100%	1	50%
UninitializedStateVariable [Error.num is never initialized. It is used in: Error.testAssert() (Desktop/SmartContractTestMeth	1	1	100%	1	100%	1	100%
StateVariableShouldBeConstant	3	3	100%	0	0%	3	100%
Line length must be no more than 80 but current length is 88	1	1	100%	1	100%	1	100%
The argument _func of the getSelector function should be sanitized before it is used in the keccak256 function to prevent p	1	0	0%	1	100%	1	100%
UsingDeprecatedContractMember	1	1	100%	0	0%	1	100%
SendsEthToArbitraryUser	1	1	100%	0	0%	1	100%
UninitializedStateVariable	1	1	100%	0	0%	1	100%
Defining constructors as functions with the same name as the contract is deprecated. Use constructor(...) { ... } instead.	1	0	0%	0	0%	1	100%
Using contract member balance inherited from the address type is deprecated. Convert the contract to address type to access the member, for example use address(contract).balance instead.	1	1	100%	0	0%	1	100%
Warning: Integer Underflow.	1	1	100%	1	100%	1	100%
Visibility modifier must be first in list of modifiers	1	1	100%	0	0%	1	100%
Warning: Defining constructors as functions with the same name as the contract is deprecated. Use constructor(...) { ... } instead.	1	0	0%	0	0%	0	0%
MissingPragmaVersion	4	3	75%	3	75%	4	100%
StateMutabilityCanBeRestricted	3	1	33%	2	67%	2	67%
Method arguments must be sanitized before they are used in computations.	4	1	25%	2	50%	4	100%
SPDX license identifier not provided in source file	1	0	0%	1	100%	1	100%
Function state mutability can be restricted to view	3	2	67%	2	67%	2	67%
ArrayLengthWithUserControlledValue	1	1	100%	0	0%	1	100%
WeakPRNG	1	0	0%	0	0%	1	100%
Reentrancy	1	1	100%	0	0%	1	100%
Indexed access to non-indexed array	1	0	0%	0	0%	1	100%
Missing constructor identifier	1	0	0%	0	0%	1	100%
ParserError: Expected identifier, got 'LParen'	1	1	100%	0	0%	1	100%
Fallback function must be simple	1	1	100%	0	0%	1	100%
Function visibility should be explicitly specified. (functions: withdraw(), payable())	1	0	0%	1	100%	1	100%
IncorrectModifier	1	1	100%	1	100%	0	0%
NonMixedCaseVariable	1	1	100%	1	100%	0	0%
NonMixedCaseParameter	4	3	75%	4	100%	2	50%
Function upgrade() calls setCompleted() on upgraded contract without checking that upgraded is not null.	1	0	0%	1	100%	0	0%
DangerousTimestampComparison	2	0	0%	2	100%	1	50%
Visibility of state variables should be stated explicitly.	3	0	0%	3	100%	3	100%
State variables should be explicitly initialized.	2	0	0%	2	100%	1	50%
MissingEventAccessControl	2	2	100%	2	100%	1	50%
MissingZeroAddressValidation	3	2	67%	2	67%	3	100%
Wrong argument count for function call: 2 arguments given but expected 1. (Wywołanie funkcji wymaga dwóch argumentów, ale oczekiwany jest jeden.)	1	0	0%	1	100%	1	100%
Avoid complex solidity version pragma statements.	2	2	100%	1	50%	1	50%
Unused state variables should be removed.	1	1	100%	1	100%	1	100%
"throw" is deprecated, avoid to use it (warning)	1	0	0%	1	100%	1	100%
DeprecatedStandard	1	0	0%	1	100%	1	100%
Code contains empty blocks	1	1	100%	1	100%	1	100%
SendsEtherToArbitraryDestination	1	0	0%	1	100%	0	0%
IncorrectERC20Interface	1	0	0%	1	100%	1	100%
LowLevelCall	1	0	0%	1	100%	0	0%
High: Unrestricted access to withdraw() function, allowing anyone to withdraw funds from the contract.	1	1	100%	1	100%	1	100%
Low: Unrestricted access to transfer() function, allowing anyone to transfer funds from the contract to any arbitrary address.	1	1	100%	1	100%	1	100%
High: Unrestricted access to transfer() function, allowing anyone to transfer funds from the contract to any arbitrary address.	1	1	100%	1	100%	1	100%
DangerousStrictEquality	1	1	100%	1	100%	1	100%
Expected identifier, got 'LParen'	1	1	100%	1	100%	1	100%
Dangerous strict equalities that use account's balance, timestamps and block numbers should be avoided.	1	1	100%	1	100%	1	100%
Line length must be no more than 80 but current length is 97 (max-line-length)	1	0	0%	0	0%	0	0%
Line length must be no more than 80 but current length is 87 (max-line-length)	1	0	0%	0	0%	0	0%
Line length must be no more than 80 but current length is 101 (max-line-length)	1	0	0%	0	0%	0	0%
Constant name must be in capitalized SNAKE_CASE	1	0	0%	0	0%	1	100%
NonMixedCaseParam	2	0	0%	2	100%	1	50%
mapping(address => Todo[maxAmountOfTodos]) public todos	1	0	0%	1	100%	1	100%
require(!todos[msg.sender][_todoId].isCompleted);	1	0	0%	1	100%	1	100%
Line length must be no more than 80 but current length is 85	1	1	100%	0	0%	1	100%
Line length must be no more than 80 but current length is 99	1	1	100%	0	0%	1	100%
PragmaVersion	2	1	50%	2	100%	2	100%
StateVarShouldBeConstant	1	0	0%	1	100%	0	0%
A public function that is never called within the contract should be marked as external.	1	1	100%	1	100%	1	100%
Method arguments must be sanitized before they are used in computations.	1	1	100%	0	0%	0	0%

Fig. 1. Summary of experiments with static analysis frameworks and LLM models

the planned tool should undoubtedly exceed the analytical and verification capabilities of the average user. For advanced users, it can also serve as a useful support, allowing them to compare their analyses with those performed by LLMs.

These initial, relatively simple experiments involved using selected LLMs to identify errors and vulnerabilities, yielding interesting results. Future experiments should expand this approach. Firstly, they ought to consider the different functionalities and goals of available frameworks, as they target the various types of errors. They ought to establish a strategy for sequentially invoking frameworks, prioritizing syntax errors, semantics, overflow vulnerabilities, gas usage threats, etc., to reduce initial errors and expect fewer in subsequent invocations. Additionally, they should use another layer of LLMs to eliminate false positives and focus on known vulnerability catalogues for targeted testing.

Just as we conducted experiments for static analysis frameworks (Figure 1), we similarly conducted them for dynamic analysis frameworks. The results of these experiments will be omitted here. Another approach could involve LLMs to generate hints for test generation for dynamic analysis frameworks. Language models can analyse source codes, extract relevant information and suggest test cases based on understanding its structure and behaviour. These hints can assist developers in identifying potential test cases to be included during software testing. Queries can be general or focused on catalogued vulnerabilities.

Here are some examples of how LLMs can provide hints for test generation:

- Boundary Value Analysis – LLMs can identify boundary conditions in code snippets and suggest test cases that cover these boundaries. For example, if a function accepts integers between 1 and 100, the LLM may suggest test cases for values like 1, 100, and values just outside this range;
- Branch Coverage – LLMs can analyse conditional statements and suggest test cases to achieve branch coverage. If-else statements, switch cases and loops with conditions can all be analysed to suggest relevant test cases that cover the different branches of the code;
- Function Invocation – LLMs can suggest test cases to ensure that functions or methods are invoked correctly with appropriate parameters. For instance, if a function expects specific input types, the LLM may suggest test cases with the different combinations of inputs;
- Integration Testing – LLMs can analyse code interactions between different modules or components and suggest test cases to verify integration points. This ensures that components work together as expected.

This list can certainly be expanded. We conducted initial experiments with LLMs Llama, Cohere and TextCortext, however, we only utilised the Solhint framework. The initial results are promising.

4. Conclusions

Throughout our experimentation with smart contract codes and LLMs, several insights have emerged. One notable challenge is the limited training data (in Solidity) available for LLMs, which impacts their performance. Despite this constraint, the results yielded intriguing findings, showing a promise across various aspects. However, it is evident that further extensive efforts are required to fully harness the potential of LLMs in smart contract development.

References

[1] He, Z., Li, Z., Yang, S., Qiao, A., Zhang, X., Luo, X., Chen, T.: Large language models for blockchain security: A systematic literature review. arXiv (2024), https://arxiv.org/abs/2403.14280

[2] Ma, W., Wu, D., Sun, Y., Wang, T., Liu, S., Zhang, J., Xue, Y., Liu, Y.: Combining fine-tuning and llm-based agents for intuitive smart contract auditing with justifications. arXiv (2024), https://arxiv.org/abs/2403.16073

[3] Shou, C., Liu, J., Lu, D., Sen, K.: Llm4fuzz: Guided fuzzing of smart contracts with large language models. arXiv (2024). https://doi.org/10.48550/ARXIV.2401.11108, https://doi.org/10.48550/arXiv.2401.11108

[4] Zhao, J., Chen, X., Yang, G., Shen, Y.: Automatic smart contract comment generation via large language models and in-context learning. Information and Software Technology **168**, 107405 (2024). https://doi.org/10.1016/j.infsof.2024.107405