Finding Vulnerabilities in Solidity Smart Contracts with In-Context Learning

Badaruddin Chachar[1,*], Marc Cavazza[2], Andrea Bracciali[3], Pietro Ferrara[1] and Agostino Cortesi[1]

[1]Ca'Foscari University of Venice, Italy
[2]University of Stirling, Scotland
[3]University of Turin, Italy

Abstract

Smart Contracts are at the heart of blockchain transactions, and their integrity is essential to blockchain reliability and performance. Specific errors in Smart Contract code are known to trigger vulnerabilities, which have been categorized into different patterns. Following the success of Large Language Models in software analysis, several authors have proposed the use of LLM to detect Smart Contract vulnerabilities. In this paper, we compare the performance of various LLM as well as formal analysis tools in analyzing Ethereum Smart Contracts for various vulnerabilities, based on standard datasets. Unlike previous work that used LLM fine-tuning, we explore performance based on direct In-Context Learning using standard Prompt Engineering techniques. Our results suggest that the straightforward use of LLM may still be beneficial in the analysis of Smart Contracts, depending on the vulnerability type.

Keywords

Large Language Models, Solidity, Smart Contract, Vulnerability Detection

1. Introduction

Ethereum is one of the blockchain platforms with the highest total value above $50 Billion[1] at the moment of writing. Smart contracts are scripts that can be deployed on Ethereum, and the blockchain guarantees their unalterable execution on predetermined conditions without needing a third party. Smart contracts are written for various application domains called Decentralized Applications (DApps). Different DApps can interact with one another [1]. Smart contracts may contain errors that can be exploited for financial gain or lead to the loss of financial assets. To prevent the loss of assets due to hacks or errors in the code, smart contracts are tested thoroughly to find and correct vulnerabilities before they are deployed on the blockchain, where they cannot be altered anymore. The vulnerability detection task involves techniques like formal verification, symbolic execution, fuzzing, intermediate representation, and machine/deep learning [2]. Since the launch of ChatGPT, the most famous Large Language Model (LLM) by OpenAI, and its efficiency on natural language tasks, studies have also been conducted to apply LLMs to code-related tasks.

In contrast to traditional web applications, where bugs can cause the unavailability of desired services, the bugs in decentralized web applications are more harmful as exploiting these bugs can result in the theft of the money managed by those applications. In the past, there have been numerous such incidents where bugs in the application were exploited to steal money, and one such example is the famous DAO attack [3], which occurred in May 2016 and resulted in the loss of $150M due to a reentrancy attack. These attacks have never stopped occurring. According to Immunefi[2], Decentralized Finance

Joint National Conference on Cybersecurity (ITASEC & SERICS 2025), February 03-8, 2025, Bologna, IT
*Corresponding author.
badaruddin.chachar@unive.it (B. Chachar); marc.cavazza@stir.ac.uk (M. Cavazza); andrea.bracciali@unito.it (A. Bracciali); pietro.ferrara@unive.it (P. Ferrara); cortesi@unive.it (A. Cortesi)
0009-0000-2718-1882 (B. Chachar); 0000-0001-6113-9696 (M. Cavazza); 0000-0003-1451-9260 (A. Bracciali); 0000-0002-4678-933X (P. Ferrara); 0000-0002-0946-5440 (A. Cortesi)

[1]https://www.coingecko.com/en/chains
[2]https://immunefi.com/research/

applications have lost $114M due to hacks in 34 incidents in the third quarter of 2024, and $1.7B was lost due to hacks in the Decentralized Finance applications in the year 2023.

LLMs have been used for finding vulnerabilities in general programming languages. Yao et al. [4] provided a detailed overview of the research into the beneficial/offensive applications of LLMs and discussed vulnerabilities in LLMs in their comprehensive literature review. Purba et al. [5] used LLM for detecting vulnerabilities in C/C++ programs. Zhou [6] evaluated the impact of providing different information in prompt on LLM performance. Noever [7] used GPT-4 for finding vulnerabilities in real-world applications, whereas Lee [8] created a dataset to evaluate the performance of LLM for finding syntactical errors in Python code. Guo [9] compared performances of task-specific deep learning models, generic pre-trained LLMs, and fine-tuned LLMs for vulnerability detection. In the case of general programming languages, there is a mixed opinion about the efficiency of LLM models in finding vulnerabilities in code, but the majority of studies [6, 7, 8] suggest that LLMs have better performances at detecting vulnerabilities in code than automated tools.

The development of static analysis tools based on formal methods requires a significant theoretical background and programming skills, even for designing and implementing a proof-of-concept tool, making it difficult for traditional developers and blockchain practitioners [10]. Several vulnerability detection tools have been proposed in the literature. Qian et al. [11] list 29 fully automated tools that are used to detect vulnerabilities from solidity source code and are based on formal verification, symbolic execution, fuzzing, intermediate representation, or deep learning techniques. The efficiency of these automated tools for finding novel vulnerabilities in smart contracts is not good, as Wei et al. [2] have proved that these tools can only detect a specific type of vulnerabilities, and Galeb and Pattabiraman [12] proved that automated tools sometimes can not detect the vulnerabilities that they claim they can detect. Zhang et al. [13] and Zheng et al. [14] have proved that automated tools are not effective at detecting vulnerabilities that could be found by manual audits of the source code. Unfortunately, the currently available static analyzers are hardly accompanied by formal soundness guarantees whereas the deep generative language models are gaining widespread adoption and progressively emerging as superior to static analyzers when measuring their performance in terms of both precision and speed of the analysis [15].

1.1. Research Gap and Novel Contribution

It is evident from the works of Zhang [13] and Zheng [14] that automated tools are not always as efficient as desired at detecting vulnerabilities in real-world solidity smart contracts. In contrast, Wei in [2] concluded that each automated tool is good at finding specific types of vulnerabilities in solidity smart contracts. Noever added evidence to it [7] and Lee [8] showed that LLMs can perform better than automated tools in finding bugs in general programming languages. So, having seen that LLMs performs better in finding vulnerabilities in general programming languages than automated tools, we are interested in using LLMs to find vulnerabilities in solidity smart contracts.

This paper will present an *in-context learning prompt*, check its performance in detecting vulnerabilities in solidity smart contracts using different LLMs, and compare it to static analysis tools and state-of-the-art fine-tuned models. Extensive experimental results suggest that even in the absence of fine-tuning, straightforward use of LLMs may still be beneficial for vulnerability detection in the analysis of smart contracts.

The rest of the paper is organized as follows. Section 2 gives a detailed overview of the literature, Section 3 describes the vulnerabilities in Solidity smart contracts and methodology is presented in Section 4, experiments are reported in Section 5 and conclusions are given in Section 6.

2. Literature Review

Since ChatGPT was released to the general public as the first large language model (LLM) that received the spotlight in the general public due to its ability to generate coherent text, the scientific community has

started investigating its applications in different fields, including vulnerability detection in blockchain scenarios using LLMs.

Chen et al. [16] use ChatGPT to find vulnerabilities in smart contracts and compared it with state-of-the-art vulnerability detection tools and found that ChatGPT performed slightly better for 4/7 vulnerability types. In contrast, for other 3/7 types of vulnerabilities, ChatGPT does not produce good F1-score as compared to the state of the art vulnerability detection analyzers. They compared the vulnerability detection capability of ChatGPT with other static analyzers and also listed the limitations of ChatGPT in detecting vulnerabilities in smart contracts. They found that ChatGPT could provide different vulnerabilities for the same smart contracts with the same prompt. Specifically, they tested 50 smart contracts each five times and found that for 29/50, ChatGPT gave consistent results, which means ChatGPT is not consistent in vulnerability detection for 42% of the smart contracts.

David et al. [17] used LLM for auditing 52 previously compromised smart contracts with the optimized prompt. They used GPT-4-32k and Claude-v1.3-100k LLM models for auditing. They classified 52 smart contracts into 38 vulnerability types. They prompted both models for a binary outcome (yes/no for the presence or absence of each vulnerability) for each vulnerability type against each smart contract. They got 40% accuracy, but the number of false positives is too high.

Du and Tang [18] used the ChatGPT-4 model to detect 7 types of vulnerabilities in 35 smart contracts containing 732 vulnerabilities. The model had 96.6% precision, but recall, and f1 scores are 37.8% and 41.1%, indicating that the model does not perform well.

In GPTScan, Sun et al. [19] combined ChatGPT with a static analysis tool. It used static analysis tools to detect candidate functions, which are then processed with the GPT module to find key statements or variables. These key variables/statements are passed to static analysis tools with vulnerability types to confirm vulnerabilities. They used three datasets. First, Top200 are projects from top 200 market capitalization which are thoroughly audited and assumed to be free from vulnerabilities. Second, Web3Bugs consists of 72 Code4rena-audited projects. Third, DefiHacks consists of 13 projects that have been previously exploited. GPTScan doesn't produce significant false positives and has better precision and recall than static analysis or other GPT-based tools. Their static confirmation further reduces the false positive cases in the Web3Bugs dataset. This tool costs 0.01 USD to analyze a thousand lines of solidity code in 14.39 seconds, and effectively found nine new vulnerabilities in the Web3Bugs dataset that were missed in manual audits. So, in conclusion, this tool can be useful to human auditors so that they can perform their jobs effectively.

Regarding the efficiency of LLMs for finding vulnerabilities in solidity smart contracts, [16] finds the GPT results inconsistent between different executions and [17, 18] report a high false positive rate. On the other hand, [19] reports that GPT does not produce false positives and even finds bugs that are not caught by automated tools or manual analysis.

3. Background

3.1. Ethereum Platform

Ethereum is a globally decentralized blockchain that supports smart contract functionalities. Usually, Ethereum smart contracts are written in Solidity, the very first programming language for smart contracts. This programming language is Turing-complete, so it can encode very complex functionalities and interact with data from different sources. The Ethereum Virtual Machine (EVM) is the platform that executes these smart contracts.

3.2. Smart Contract Vulnerabilities

Several studies have grouped the vulnerabilities for more effortless organization and reducing redundancy. A comprehensive literature survey between 2018 and 2023 reported the most common vulnerabilities [20], and by considering their actual impact when exploited, we will focus on the following three.

3.2.1. Reentrancy

This vulnerability can occur when any user's balance is updated after transferring the balance using *msg.sender*. Using this vulnerability an attacker can withdraw the balance multiple times by recursively calling the withdraw function. In the following example in Listing 1, the user's balance is checked on line 9, and the amount is sent on line 11, whereas the balance is updated after the transfer on line 12. If an attacker keeps calling the withdraw function recursively from its fallback function, he can withdraw the amount until draining the balance of the contract completely as the balance is updated after transfer. The example code is from [21].

```
1  contract SimpleDAO {
2    mapping (address => uint) public credit;
3
4    function donate(address to) payable {
5      credit[to] += msg.value;
6    }
7
8    function withdraw(uint amount) {
9      if (credit[msg.sender]>= amount) {
10       // Vulnerable: reentrancy
11       bool res = msg.sender.call.value(amount)();
12       credit[msg.sender]-=amount;
13     }
14   }
15
16   function queryCredit(address to) returns (uint){
17     return credit[to];
18   }
19 }
```

Listing 1: Reentrancy Code Example

3.2.2. Timestamp Manipulation

Timestamp dependency vulnerability occurs when *block.timestamp* is used as a condition for a critical operation like money transfer or as a random number seed. Using this *block.timestamp* miners can schedule the transaction at a favorable time. In the following code, *claim* function uses *block.timestamp* in line 5 as a condition to execute the function. The timestamp values are generated by the node that create the block that contain the transaction that execute the smart contract, this way the node can generate a timestamp that is favorable to them and exploit the contract. The example code in Listing 2, is from [20].

```
1  contract Vulnerable {
2    uint public prevClaimTime;
3    // Vulnerable: relies on block.timestamp.
4    function claim() public {
5      require(block.timestamp > prevClaimTime + 1, "Unable to claim yet");
6      prevClaimTime = block.timestamp;
7    }
8  }
```

Listing 2: Timestamp Manipulatin Code Example

3.2.3. Unchecked External Call

Unchecked external call vulnerability occurs when a contract calls another contract; the called contract can fail silently without throwing an exception. If the calling contract does not check the outcome, it might assume that the call was successful, even if it was not. In the following code, on line 9, *winner.send* does not check the outcome of the call and assumes that the call went successful, and the

same vulnerability occurs on line 16, where the successful execution of the call is not checked. The example code in Listing 3, is from [20].

```
contract Lotto {
  bool public payedOut = false;
  address public winner;
  uint public winAmount;

  function sendToWinner() public {
    require(!payedOut);
    // Vulnerable: unchecked external call
    winner.send(winAmount);
    payedOut = true;
  }

  function withdrawLeftOver() public {
    require(payedOut);
    // Vulnerable: unchecked external call
    msg.sender.send(this.balance);
  }
}
```

Listing 3: Unchecked External calls Code Example

4. Methodology

In this section, we outline our methodology by detailing the key steps of our study. We start with the choice of automated analyzers and LLM models. Next, we provide a breakdown of our LLM prompt design based on the In-Context Learning (ICL) approach. Finally, we describe the dataset selection process for the experimental evaluation.

4.1. Static Analyzers

Static analysis aims to infer and prove the properties of programs without executing them. Several static analyzers have been proposed, ranging from efficient but imprecise syntactic analyzers to precise, sound, but expensive semantic analyzers. Several approaches, such as abstract interpretation, program verification, and model checking, have been widely applied to different contexts. Smart contracts have also been a target of this work.

Based on the performance comparison of different automated analysis tools over different vulnerability types presented in [2], we consider two tools that are able to detect the same set of vulnerabilities (namely reentrancy, timestamp manipulation, and unchecked external calls) and have quite good performance: Conkas[22] and Slither[23]. Both have more than 90% F1 score for all the vulnerability categories we are interested in, outperforming other available tools: the F1-score for reentrancy is 98.8% and 96.9%, respectively, while for timestamp manipulation is 93% and 99% respectivery, and for unchecked external calls is 98.3% and 96.4%, respectively.

4.1.1. Conkas [22]

is a symbolic execution tool that uses Intermediate Representation. It can analyze source or byte code. It supports five categories: Arithmetic, Front-Running, Reentrancy, Time Manipulation, and Unchecked Low-Level Calls.

4.1.2. Slither [23]

is a static analysis framework that uses an abstract syntax tree generated by the Solidity compiler and converts it into an intermediate representation called SlithIR. Slither supports more types of vulnerabilities than Conkas. The Slither GitHub page lists 93 detector modules for detecting different

vulnerabilities. We have used the following modules to detect each vulnerability type, which has taken less time to finish the analysis.

- **Reentrancy:** *reentrancy-unlimited-gas, reentrancy-no-eth, reentrancy-benign, reentrancy-eth, reentrancy-events*
- **Timestamp:** *timestamp*
- **Unchecked:** *unchecked-lowlevel, unchecked-send*

4.2. LLM Models

In our research, we have selected three prominent Large Language Models (LLMs) to identify vulnerabilities in Solidity smart contracts:

- GPT-4o by OpenAI: This model was chosen for its advanced reasoning capabilities and improved performance in tasks such as language, vision, and speech processing. A comprehensive evaluation of GPT-4o's multimodal proficiency is detailed in this study. [24]
- Mistral-7B by Mistral AI: We selected this model due to its high performance in both code and English language tasks. Mistral-7B is a 7.3 billion parameter language model that utilizes Grouped-query Attention (GQA) for faster inference and Sliding Window Attention (SWA) for handling longer text sequences efficiently. Details of its architecture and performance are discussed in this paper. [25]
- Claude-Sonnet by Anthropic: This model was included for its robust natural language processing capabilities and effectiveness in reasoning about security vulnerabilities. Claude 3.5 Sonnet has been evaluated in cybersecurity contexts, including capture-the-flag challenges that test vulnerability discovery and exploit development. Insights into its performance and capabilities are provided in this study.

By leveraging these diverse models, we aim to conduct a comprehensive analysis of vulnerability detection in Solidity smart contract, benefiting from each model's unique strengths.

4.3. Prompt Engineering

Prompt Engineering is a set of techniques to enhance the effectiveness of LLMs. In-context learning (ICL) is one of these techniques that allows language models to learn tasks given only a few examples in the form of demonstration. The model learns the hidden patterns in the demonstrated examples and makes predictions accordingly. LLMs perform complex mathematical reasoning tasks with ICL. The examples of a complex task are presented in the prompt to serve as an analogy and increase the efficiency of the LLM [26].

Prompt Design: For our work, we have used ICL with vulnerability description and annotated examples given to the LLM to make the correct decision. Zhou [6] has provided a detailed study of bits of information to include in the ICL prompt and its impact on performance. The following information is given to the LLM inside the prompt.

- **Task Description** instructs the LLM to act as a smart contract auditor for a specific vulnerability type.
- **Vulnerability Description** provides a detailed description of the vulnerability in natural language; specifically, it describes how a specific vulnerability occurs inside the code.
- **Examples** are annotated with HTML tags around the lines that contain the vulnerability. We have given three example codes as Zhou [6] found that three examples effectively detect bugs.
- **Certainty Score** is required from the LLM to show how much confident the model is about its response.
- **Output Format** restricts the LLM model to produce output in JSON format only, which should contain the name of the contract and the location of vulnerability, if any detected (otherwise a *null* message).

Prompt

```
You are a Solidity smart contract auditor. Your task is to analyze a given
solidity smart contract for {vulnerability_type} vulnerabilities.
#Vulnerability Description: Description of the vulnerability
#Examples: Following three contracts examples contain {vulnerability_type}
vulnerabilities.
Example contract 1: {example1_code}
Example contract 2: {example2_code}
Example contract 3: {example3_code}
#Task The contract to audit can be found in this file, analyze this contract
thoroughly and report if it contains {vulnerability_type} vulnerabilities.
Test contract to audit is as follows: {testContract_Code}
#Certainity Score Please provide a certainty score once you have identified a
potential vulnerability, rate your level of confidence in the form of a single %
score.
#Output format The output should be a JSON object with the following format:

{"smart_contracts": [
  { "name": "SampleSmartContract1",
    "contains_vulnerability": "no",
    "vulnerability_location": null },
  { "name": "SampleSmartContract2",
    "contains_vulnerability": "yes",
    "vulnerability_location": {
      "description": "line of code that produces the vulnerability", } }
],
"certainty_score": 0
}
```

All the smart contract code examples are taken from the smartsbugs dataset [27] and annotated manually for each vulnerability type. The examples have annotations around lines of code that contain bugs. One such code example for vulnerability type reentrancy is given below in Listing 4.

```
contract SimpleDAO {
  mapping (address => uint) public credit;

  function donate(address to) payable {
    credit[to] += msg.value;
  }

  function withdraw(uint amount) {
    if (credit[msg.sender]>= amount) {
      <REENTRANCY>
      bool res = msg.sender.call.value(amount)();
      credit[msg.sender]-=amount;
      </REENTRANCY>
    }
  }

  function queryCredit(address to) returns (uint){
    return credit[to];
  }
}
```

Listing 4: Annotated Code Example

4.4. Dataset

Two of the earliest used datasets for smart contract vulnerability detection are smartbugs-curated and smartbugs-wild. The smart bugs-curated dataset [27] only contains solidity code that contains bugs,

but the number of examples per category was insufficient for our use since we were interested in classification tasks. Furthermore, we needed a dataset containing positive and negative code files. The smartbugs-wild dataset was too large for our purpose, as it contained 47,0000 solidity files and was not annotated. Similarly, smartcontract-benchmark [2] contains two datasets, a small dataset contains almost ten files with vulnerable code for each category, while a scaled dataset contains 20,000 unlabeled solidity files crawled from the web site etherscan. So, this dataset was also not usable for our purpose for the same reasons.

The dataset we selected for our experiments was published in [28]. It was crawled from etherscan, and duplicates were removed by comparing the file hash. All the vulnerabilities were annotated after a manual inspection of the code. It was annotated for eight vulnerability types, but we only used three categories. The original dataset was skewed because it contained more negative instances, i.e., files that did not contain vulnerabilities and a few cases where the contract contained vulnerabilities. So, to overcome this imbalance in the dataset, we have used part of it, as we have taken 50 positive and 50 negative smart contract files for each of three categories: reentrancy, unchecked external call, and timestamp dependency.

5. Experiments

5.1. Computing System

HP EliteBook 830 G6 was used for all the computations reported. The system specifications were Intel(R) Core(TM) i7-8565U CPU @ 1.80GHz with 16 GB RAM. The versions for different programs used were:

- OS: ArchLinux 6.6.49-1-lts
- Python: 3.9.18
- Slither: 0.10.4
- Conkas: Mar 18, 2021 release

5.2. Evaluation Metrics

For evaluation and comparison, we have used the F1-score, which is a standard evaluation technique for classification tasks. The F1-score is calculated using precision and recall, both of which depend on the confusion matrix.

5.2.1. Confusion Matrix

In machine learning classification, a confusion matrix is used to evaluate the performance of a model. It summarizes the number of correct and incorrect predictions made by the model compared to the actual values.

- **True Positive (TP):** The model correctly predicts the positive class.
- **True Negative (TN):** The model correctly predicts the negative class.
- **False Positive (FP):** The model incorrectly predicts positive when it is actually negative.
- **False Negative (FN):** The model incorrectly predicts negative when it is actually positive.

Actual / Predicted	Predicted Positive	Predicted Negative
Actual Positive	TP	FN
Actual Negative	FP	TN

Table 1
Confusion Matrix for Binary Classification

5.2.2. Accuracy

It measures the overall correctness of the model which can be calculated using this formula 1.

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN} \tag{1}$$

5.2.3. Precision

It measure the portion of correctly predicted cases among all predicted cases. Precision focuses on how many of the predicted positive cases are actually positive. The formula for calculating precision is 2

$$\text{Precision} = \frac{TP}{TP + FP} \tag{2}$$

5.2.4. Recall

It measures the proportion of correctly predicted positive cases among all actual positives. Recall measures how many actual positive cases were correctly identified by the model. Recall can be calculated using formula 3

$$\text{Recall} = \frac{TP}{TP + FN} \tag{3}$$

5.2.5. F1 Score

F1 score is a harmonic mean of precision and recall and it is treated as final evaluation standard for the classification tasks. Following formula 4 is used to calculate F1 score.

$$F1 = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} \tag{4}$$

5.3. Discussion of Results

Using our prompt, we tested three LLMs, i.e., gpt4o, mistral-7b and Claude sonnet, and results are presented in Table 2 along with two static analysis tools. The F1 score varies from category to category, and the GPT-4o model got the best F1 score, 89%, in the reentrancy category. Other models performed better than automated tools. For the Timestamp category, the GPT-4o model performed better than other models with a 69% F1 score. All the models have similar F1 scores on the Unchecked category: gpt40 got 67%, mistral-7b got 68%, and Claude got 66%.

Although both Conkas and Slither are reported to achieve more than 90% F1 scores in the selected categories, their performance varies highly. Conkas performed better on the reentrancy and Timestamp categories, whereas Slither can only detect vulnerabilities in the Timestamp category. Both of these analyzers didn't perform well in the Unchecked category. Their performance is significantly lower than that of LLMs in the reentrancy category and somewhat equal in the Timestamp category. In contrast, LLMs performed much better than automated tools in the Unchecked category.

Another study i.e FELLMVP [29] is conducted by Luo et al. using the same dataset. They have fine-tuned the Gemma-7B model for each category and then created an ensemble LLM for vulnerability detection. They have got 88% F1 score across all categories using top-3 classification, but they have also reported an increase of 75% in F1 score from top-1 to top-2 and another 19% increase in F1 score from top-2 to top-3 classification. The comparison depicted in Table 3 shows that GPT-4o with our In-Context-Learning prompt performs close to those ensembled fine-tuned models in FELLMVP for the reentrancy category and for the other two categories, the performance of gpt4o is not far below that of the fine-tuned models but this can be improved by further research into improving the prompt for these two categories with more suitable examples or description of the vulnerability.

Category	Analyzer	Metrics			
		Accuracy	Precision	Recall	F1
Reentrancy	gpt4o	80.6	100.0	88.0	89.3
	Mixtral	64.1	100.0	72.0	78.1
	Claude	76.9	100.0	85.0	87.0
	conkas	56.2	100.0	61.0	71.9
	slither	5.9	2.0	35.0	3.0
Timestamp	gpt4o	54.5	96.0	58.0	69.6
	Mixtral	44.6	66.0	42.0	53.2
	Claude	49.4	82.0	49.0	61.7
	conkas	66.0	66.0	66.0	66.0
	slither	59.2	84.0	63.0	69.4
Unchecked	gpt4o	50.5	98.0	51.0	66.7
	Mixtral	58.8	80.0	62.0	67.8
	Claude	49.5	98.0	49.0	65.8
	conkas	44.4	8.0	49.0	13.6
	slither	57.1	8.0	51.0	14.0

Table 2
Vulnerability Categories with Metrics

	Reentrancy	Timestamp	Unchecked
FELLMVP	95.1	86.1	94.1
Our Best	89.3	69.9	66.7

Table 3
F1-Score Comparison with FELLMVP

5.4. Limitations

- LLMs are evolving rapidly, and the performance of future versions of the LLMs used in the study, or the release of more software-specialized LLM may significantly alter the reported rankings.
- Despite setting completion parameters (e.g., temperature or top_p) to control randomness, the inherent variability in the model's responses remains. This variability can lead to different outputs across runs, potentially impacting consistency and reliability. While we acknowledge this factor, we have not systematically addressed it through repeated runs and statistical analysis. Incorporating such an approach in future work could provide deeper insights into the stability of results, helping to quantify uncertainty and refine decision-making based on LLM outputs.
- These results depend on the selected datasets which are a good reflection of the state-of-the-art but may evolve as well with the smart contract technology.

6. Conclusion

While the state-of-the-art in LLM-based smart contract analysis has been established by systems relying on application fine-tuning, we wanted to explore how simpler in-context learning could achieve a balance between ease of development and performance. To that effect, we have explored prompting techniques to detect three types of vulnerabilities in solidity smart contracts, using different LLMs. We found that performance varies both across vulnerability categories and across LLMs. The GPT-4o model we used performed best among others while also outperforming static analysis tools (on F1 score). For

the specific vulnerability of reentrancy, our model performs within 6% as the state-of-the-art model, which can be advantageous considering the immediate readiness of a Prompt-based approach and the need for human supervision in all cases. In comparison to fine-tuning an LLM, our strategy of Prompt Engineering using In-Context Learning can be easily scaled up to different categories of vulnerabilities in Solidity smart contracts and can also be adapted to other programming languages.

Acknowledgments

- We would like to acknowledge **Rabeea Abid**, a Master's student at the University of Stirling, Scotland, for her valuable contribution in conducting some of the experiments for this research. Her efforts and assistance have been greatly appreciated.
- This work was partially supported by the SERICS (PE00000014 - CUP H73C2200089001) and iNEST (ECS00000043 – CUP H43C22000540006) projects funded by PNRR NextGeneration EU.

Declaration on Generative AI

- During the preparation of this work, the authors used AI-based tools (*Grammarly*, *ChatGPT*) in order to: Grammar and spelling check, Paraphrase and reword. After using this tool/service, the authors reviewed and edited the content as needed and take full responsibility for the publication's content.
- In the experimental evaluation, three Generative AI models (*GPT-4o by OpenAI*, *Mistral-7B by Mistral AI*, and *Claude-Sonnet by Anthropic*) were used for the detection of smart contract issues.

References

[1] W. Metcalfe, et al., Ethereum, smart contracts, dapps, Blockchain and Crypt Currency 77 (2020) 77–93.

[2] Z. Wei, J. Sun, Z. Zhang, X. Zhang, M. Li, L. Zhu, A comparative evaluation of automated analysis tools for solidity smart contracts, arXiv preprint arXiv:2310.20212 (2023).

[3] S. Falkon, The Story of the DAO — Its History and Consequences, https://medium.com/swlh/the-story-of-the-dao-its-history-and-consequences-71e6a8a551ee, 2017. Accessed: 2024-10-28.

[4] Y. Yao, J. Duan, K. Xu, Y. Cai, Z. Sun, Y. Zhang, A survey on large language model (llm) security and privacy: The good, the bad, and the ugly, High-Confidence Computing 4 (2024) 100211. URL: https://www.sciencedirect.com/science/article/pii/S266729522400014X. doi:https://doi.org/10.1016/j.hcc.2024.100211.

[5] M. D. Purba, A. Ghosh, B. J. Radford, B. Chu, Software vulnerability detection using large language models, in: 2023 IEEE 34th International Symposium on Software Reliability Engineering Workshops (ISSREW), IEEE, 2023, pp. 112–119.

[6] X. Zhou, T. Zhang, D. Lo, Large language model for vulnerability detection: Emerging results and future directions, in: Proceedings of the 2024 ACM/IEEE 44th International Conference on Software Engineering: New Ideas and Emerging Results, 2024, pp. 47–51.

[7] D. Noever, Can large language models find and fix vulnerable software?, arXiv preprint arXiv:2308.10345 (2023).

[8] H. Lee, S. Sharma, B. Hu, Bug in the code stack: Can llms find bugs in large python code stacks, arXiv preprint arXiv:2406.15325 (2024).

[9] Y. Guo, C. Patsakis, Q. Hu, Q. Tang, F. Casino, Outside the comfort zone: Analysing llm capabilities in software vulnerability detection, arXiv preprint arXiv:2408.16400 (2024).

[10] L. Olivieri, F. Spoto, Software verification challenges in the blockchain ecosystem, International Journal on Software Tools for Technology Transfer 26 (2024) 431–444. URL: https://doi.org/10.1007/s10009-024-00758-x. doi:10.1007/s10009-024-00758-x.

[11] P. Qian, Z. Liu, Q. He, B. Huang, D. Tian, X. Wang, Smart contract vulnerability detection technique: A survey, arXiv preprint arXiv:2209.05872 (2022).
[12] A. Ghaleb, K. Pattabiraman, How effective are smart contract analysis tools? evaluating smart contract static analysis tools using bug injection, in: Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, 2020, pp. 415–427.
[13] Z. Zhang, B. Zhang, W. Xu, Z. Lin, Demystifying exploitable bugs in smart contracts, in: 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE), IEEE, 2023, pp. 615–627.
[14] Z. Zheng, J. Su, J. Chen, D. Lo, Z. Zhong, M. Ye, Dappscan: building large-scale datasets for smart contract weaknesses in dapp projects, IEEE Transactions on Software Engineering (2024).
[15] D. Ressi, A. Spanò, L. Benetollo, C. Piazza, M. Bugliesi, S. Rossi, Vulnerability detection in ethereum smart contracts via machine learning: A qualitative analysis, arXiv preprint arXiv:2407.18639 (2024).
[16] C. Chen, J. Su, J. Chen, Y. Wang, T. Bi, Y. Wang, X. Lin, T. Chen, Z. Zheng, When chatgpt meets smart contract vulnerability detection: How far are we?, arXiv preprint arXiv:2309.05520 (2023).
[17] I. David, L. Zhou, K. Qin, D. Song, L. Cavallaro, A. Gervais, Do you still need a manual smart contract audit?, arXiv preprint arXiv:2306.12338 (2023).
[18] Y. Du, X. Tang, Evaluation of chatgpt's smart contract auditing capabilities based on chain of thought, arXiv preprint arXiv:2402.12023 (2024).
[19] Y. Sun, D. Wu, Y. Xue, H. Liu, H. Wang, Z. Xu, X. Xie, 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.
[20] C. De Baets, B. Suleiman, A. Chitizadeh, I. Razzak, Vulnerability detection in smart contracts: A comprehensive survey, arXiv preprint arXiv:2407.07922 (2024).
[21] F. R. Vidal, N. Ivaki, N. Laranjeiro, Openscv: an open hierarchical taxonomy for smart contract vulnerabilities, Empirical Software Engineering 29 (2024) 101.
[22] N. Veloso, Conkas: A modular and static analysis tool for ethereum bytecode, 2023.
[23] J. Feist, G. Grieco, A. Groce, Slither: a static analysis framework for smart contracts, in: 2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB), IEEE, 2019, pp. 8–15.
[24] S. Shahriar, B. Lund, N. R. Mannuru, M. A. Arshad, K. Hayawi, R. V. K. Bevara, A. Mannuru, L. Batool, Putting gpt-4o to the sword: A comprehensive evaluation of language, vision, speech, and multimodal proficiency, 2024. URL: https://arxiv.org/abs/2407.09519. arXiv:2407.09519.
[25] A. Q. Jiang, A. Sablayrolles, A. Mensch, C. Bamford, D. S. Chaplot, D. de las Casas, F. Bressand, G. Lengyel, G. Lample, L. Saulnier, L. R. Lavaud, M.-A. Lachaux, P. Stock, T. L. Scao, T. Lavril, T. Wang, T. Lacroix, W. E. Sayed, Mistral 7b, 2023. URL: https://arxiv.org/abs/2310.06825. arXiv:2310.06825.
[26] Q. Dong, L. Li, D. Dai, C. Zheng, Z. Wu, B. Chang, X. Sun, J. Xu, Z. Sui, A survey on in-context learning, arXiv preprint arXiv:2301.00234 (2022).
[27] J. F. Ferreira, P. Cruz, T. Durieux, R. Abreu, Smartbugs: A framework to analyze solidity smart contracts, in: Proceedings of the 35th IEEE/ACM international conference on automated software engineering, 2020, pp. 1349–1352.
[28] Z. Liu, P. Qian, J. Yang, L. Liu, X. Xu, Q. He, X. Zhang, Rethinking smart contract fuzzing: Fuzzing with invocation ordering and important branch revisiting, IEEE Transactions on Information Forensics and Security 18 (2023) 1237–1251.
[29] Y. Luo, W. Xu, K. Andersson, M. S. Hossain, D. Xu, Fellmvp: An ensemble llm framework for classifying smart contract vulnerabilities, in: 2024 IEEE International Conference on Blockchain (Blockchain), IEEE, 2024, pp. 89–96.