Journal of Artificial Intelligence and Technology, 2025, 5, 314-322
https://doi.org/10.37965/jait.2025.0811

ISTP
RESEARCH ARTICLE

Three Large Language Models for Solidity Smart Contract Vulnerability Detection

Amjad Almaghthawi,[1] Wael M. S. Yafooz,[1] and Nasser S. Albalawi[2]
[1]Faculty of Computer Science Department, College of Computer Science and Engineering, Taibah University, Madinah, Saudi Arabia
[2]Department of Computer Science, Faculty of Computing and Information Technology, Northern Border University, Arar, Saudi Arabia

(Received 08 May 2025; Revised 26 June 2025; Accepted 28 July 2025; Published online 20 August 2025)

Abstract: In decentralized apps, smart contracts are used to conduct trusted transactions on the Blockchain (BC). While smart contracts are highly effective, they are also highly susceptible to security flaws, leading to serious financial consequences. However, the combination of BC technology and artificial intelligence provides a solution for powerful, secure, and decentralized applications in various sectors. Furthermore, large language models (LLMs), which are essential advanced machine learning frameworks, are now used in various applications, including customer service, chatbots, code generation, vulnerability detection, and language translation.
This study investigates the use of LLMs for automated vulnerability detection in Solidity-based smart contracts. Specifically, three models are evaluated and compared: GPT-3.5-turbo, DeepSeek R1, and LLaMA-3. With a labeled, multi-class dataset including four vulnerability types, the models are assessed across three reasoning strategies: zero-shot, few-shot, and chain of thought. A prompt-based evaluation and performance comparison is conducted using standard metrics such as accuracy, precision, recall, F1-score, and average detection time.
Results show that in the zero-shot setting, GPT-3.5-turbo achieves the highest accuracy of 94.59%, followed closely by LLaMA-3 with 92%, while DeepSeek R1 achieved 78.95%. In the few-shot setting, LLaMA-3 outperformed other models. Furthermore, in the CoT setting, LLaMA-3 demonstrates the strongest overall performance with 96% accuracy and an F1-score of 0.82, surpassing DeepSeek R1's average of 78.95% and GPT-3.5's CoT performance, which is notably lower. Hence, this study develops an evaluation framework for LLM-based vulnerability detection, and we have demonstrated that prompt engineering has the potential to enhance the security of smart contracts.

Keywords: Code Llama; DeepSeek; GPT-3.5; large language models; prompt engineering; smart contract; solidity; vulnerability detection

I. INTRODUCTION

Over the years, technology has advanced greatly. The development of Blockchain (BC) platforms such as Ethereum has led to the widespread adoption of smart contracts in many different industries. In spite of this, smart contracts are often used when handling significant financial assets or sensitive data, making them vulnerable to being hacked, exploited, and irreversible [1]. It is therefore possible to make users lose funds in addition to affecting the entire BC ecosystem. Hence, detecting vulnerabilities in smart contracts has become increasingly significant.

There have been a number of large language models (LLMs) available that have been trained on large corpora and have demonstrated remarkable performance in a variety of natural language and software engineering tasks [2]. In recent research, LLMs have gained significant attention for their potential in computer programming. There have been some recent studies on LLM's applicability toward detecting Solidity smart contract vulnerabilities. Due to the fact that GPT was one of the first LLMs available commercially, it became widely used to assess the effectiveness of the models in detecting vulnerabilities [3,4] and automatic code generation [5,6].

To our knowledge, there has been no empirical study comparing various LLMs despite a growing interest in utilizing them for vulnerability detection. This study focuses on exploring various LLMs including GPT-3.5 by the OpenAI team, CodeLlama by Meta, and DeepSeek by DeepSeek, which is a Chinese AI startup, in terms of their ability to detect vulnerabilities in the Solidity smart contracts. Moreover, LLMs can be fine-tuned or instructed with specific instructions to focus on particular categories of vulnerabilities, enhancing their efficiency when detecting BC-specific vulnerabilities. Consequently, by using customization, LLMs can improve their precision and adapt to smart contract patterns, which makes them more reliable and efficient for smart contract auditing.

However, the ability of LLMs to understand and analyze complex code patterns and logic has given them significant capability in detecting vulnerabilities. An LLM's ability to understand the relationships between different code components enables them to spot vulnerabilities that include reentrancy, access control flaws, and arithmetic overflows with great accuracy. Moreover, LLMs are capable of performing more detailed code reviews, improving

Corresponding author: Amjad Almaghthawi (e-mail: tu4570384@taibahu.edu.sa).

overall detection rates of critical vulnerabilities, and making them a promising tool in securing the BC ecosystem.

This study compares the performance of three LLMs: GPT-3.5, CodeLlama, and DeepSeek on detecting solidity smart contracts with regard to their vulnerability detection capabilities. Also, it aims to determine the most efficient LLM for smart contract auditing. The study intends to contribute toward the development of more secure BC environments and guide future advancements in artificial intelligence (AI)-driven vulnerability detection. These models might be more accurate at detecting Solidity-specific issues if they are tuned using domain-specific data, which are Solidity smart contract vulnerabilities. A model's performance can also be impacted by approaches such as zero-shot, few-shot, and chain of thought (CoT), which allow them to operate without prior examples or rely on a few examples for guidance or break down complex logic into sequential steps.

The rest of the paper is organized as follows:

Section II discusses existing approaches to detecting smart contract vulnerabilities, including traditional tools and LLM-based approaches. Then Section III includes the datasets used, prompt formats for each model, and evaluation metrics used. After that, Section IV describes experimental settings, including the fine-tuning of configurations and prompt strategies across all LLMs. Next, Section V presents results and discussion, highlighting the performance of each reasoning strategy. Section VI provides a comparative analysis, emphasizing the strengths and weaknesses of the evaluated LLMs. Lastly, Section VII discusses limitations and future directions of the study.

II. RELATED WORK

Static analysis examines the code without executing it, focusing on vulnerabilities like reentrancy and access control weaknesses. Tools such as Mythril and Slither are used for this purpose, while dynamic analysis tests behavior in a simulated environment such as Echidna and Manticore [7]. Li *et al.* [8] propose a symbolic execution-based method for detecting vulnerabilities in Ethereum smart contracts, focused on arbitrary modifications by owners. This method addresses issues like limited code coverage and time consumption by exploring all possible execution paths and states. Yashavant *et al.* highlight the lack of standardized datasets for evaluating tools for detecting vulnerabilities in Ethereum smart contracts. They created ScrawlD as an unbiased benchmark for evaluating existing and new tools [9].

In Qian *et al.* study, they examine smart contract vulnerability detection techniques, focusing on three levels: Solidity code layer, Ethereum virtual machine execution layer, and block dependency layer. They compare accuracy, F1-score, and training time using 300 real-world Ethereum smart contracts [10]. Choi *et al.* implemented SMARTIAN to enhance smart contract fuzzing using static and dynamic data-flow analyses. SMARTIAN found 211 more bugs than traditional tools [11]. Songsom *et al.* introduced SWAT, a static analysis tool that targets six smart contract weakness classification (SWC) vulnerabilities in smart contracts. SWAT is 44.58% more memory efficient than existing tools such as Slither, Mythril, and Vandal [12].

ML involves training and inference algorithms. During training, model parameters are optimized, while inference uses unlearned data to infer features. In DL, features are extracted using neural networks in a black-box manner, which is most popular today [13]. Liu *et al.* [14] suggested the transfer learning-based smart contract generation (TLSCG) framework, which outperforms existing tools by up to 18% in anomalous contract detection. Also, Boxin [15] presents a dynamic vulnerability detection approach that utilizes the N-gram model and the weight penalty mechanism for feature extraction.

Han *et al.* [16] present an ML model for detecting vulnerabilities in Ethereum smart contracts using control flow graphs and GNNs to learn semantic and structural features associated with vulnerabilities. Zhang *et al.* [17] use a Bi-LSTM neural network to detect vulnerabilities in smart contracts, achieving a high detection rate of around 80% for various vulnerabilities but improving recall performance. Also, Yang *et al.* present method based on an abstract syntax tree (AST) for detecting vulnerabilities in smart contracts. In this method, AST-based representations are used to automatically learn and detect vulnerabilities, resulting in high accuracy and effectiveness [18].

OpenAI's GPT3.5 and GPT-3 language models offer fluency, coherence, and contextual understanding for various applications [19]. Hu *et al.* propose an adversarial framework called GPTLENS to detect vulnerabilities in smart contracts using LLMs, balancing the generation of vulnerabilities and minimizing false positives. This framework improves smart contract auditing significantly without requiring specialized expertise in smart contracts [20]. According to Chen *et al.* examine ChatGPT's ability to identify smart contract vulnerabilities and compare it with other tools. However, ChatGPT showed promise in detecting some vulnerabilities but struggled with precision and displayed false positives [4].

Also, Gao *et al.* study ChatGPT's capabilities of identifying machine unauditable bugs (MUBs) within smart contracts and compared it with SPCON [21]. In another study, Boi *et al.* developed VulnHunt-GPT to detect vulnerabilities in smart contracts using OpenAI's GPT-3 model. VulnHunt-GPT achieved broad coverage and successfully detected 128 vulnerabilities in smart contracts [22]. Du *et al.* evaluated GPT-4's performance in auditing smart contracts for vulnerabilities and found mixed results when generating proof of concept exploits [23].

Furthermore, DeepSeek-V3 and DeepSeek-R1 are open-source LLMs that provide SOA performance at a lower training cost [24]. Karanjai *et al.* introduced Smartify, a multi-agent LLM framework for vulnerability detection and repair. DeepSeek models performed poorly in Solidity and Move languages, but Smartify consistently outperformed both models in accuracy, vulnerability coverage, and inference efficiency [25].

The CodeLlama LLM family is optimized for code generation and infilling and uses binary classification and multi-class classification to detect vulnerabilities in smart contracts written in Solidity using LLMs [26]. The SmartVD vulnerability detection framework, developed by Alam *et al.* [27], uses binary and multi-class classification to identify vulnerabilities in Solidity smart contracts. It uses the VulSmart dataset, with GPT-3.5 and CodeLlama achieving the best scores. However, CodeLlama struggles with complex adversarial cases. Zhang *et al.* explore the effectiveness of LLMs in automated vulnerability localization (AVL). It demonstrates that discriminative fine-tuning outperforms existing learning-based methods and introduces remedial strategies for better performance [28]. Another LLM evaluation presented by Xiao *et al.* [29] shows that a well-designed prompt can reduce false-positive rates by over 60%, and recall rates drop to just 13%. Tables I and II demonstrate a comparison between this study and some of the existing studies.

Table I. Comparison between this study and some of the existing studies

Study	Model	Accuracy	Prompting strategies	Best performing prompt
Li *et al.*	Symbolic execution system	92.3%	N/A	EVM-level opcode extraction
Liu *et al.*	Transfer learning + Generation	72%	N/A	Generative contract variants
Boxin	N-gram + ML (dynamic)	93.71%	N/A	Sequence-based opcode analysis
Han *et al.*	GNN + CNN	94.33%	N/A	N/A
Zhang *et al.*	Bi-LSTM	80%	N/A	Code-to-vector sequence model
Hu *et al.*	GPT-4-based LLM	76.9%	Strategic CoT	AUDITOR + CRITIC prompting
Chen *et al.*	GPT-3.5/GPT-4	88.2%	General prompting	CoT
Gao *et al.*	GPT-3.5/GPT-4	33%	Guidance prompting	Human-like clarification

Table II. Comparison between this study and some of the existing studies continuous

Study	Model	Accuracy	Prompting strategies	Best performing prompt
Alam *et al.*	LLMs (CodeLlama, LLaMA2, CodeT5, Falcon, GPT-3.5, and GPT-4o)	99% detection, 94% type identification, and 98% severity.	Zero-shot, few-shot, CoT	Zero-shot, CoT
This study	Fine-tuned LLMs (Gpt-3.5-turbo, DeepSeek-R1, LLaMA-3)	96%	Zero-shot, few-shot, CoT	CoT

III. METHODOLOGY

The study uses open datasets from Github to classify Solidity smart contract code into four types of vulnerabilities. Preprocessing steps are performed to remove comments, normalize whitespace, and handle identifiers. Syntax highlighting and tokenization are used to break down the code into smaller units for analysis. Fine-tuning involves training an LLM on a specialized dataset to adapt to specific tasks.

The models GPT-3.5, DeepSeek, and LLaMA are then fine-tuned with input–output pairs and associated vulnerabilities fed to the model. Figure 1 presents experimental workflows designed to evaluate prompting strategies, with zero-shot prompting providing task-specific instructions, few-shot prompting providing labeled examples of vulnerabilities, and CoT prompting instructing the model to analyze each contract function for specific vulnerabilities [30].

A. Model Architecture Overview

There are three transformer models used in this study, GPT-3.5-Turbo, LLaMA-3, and DeepSeek R1, which have the same basic transformer architecture; however, their scale, optimization objectives, and access models are different from each other.

GPT-3.5-Turbo, developed by OpenAI, is a closed-source model that was trained on massive amounts of internet data and optimized for interactivity in the form of chat. This system is designed to handle general-purpose tasks by means of prompt-based learning. While its architecture isn't publicly disclosed, it uses instruction tuning and reinforcement learning from human feedback (RLHF) to improve output quality.

LLaMA-3 is a decoder-only transformer model from Meta AI that enables researchers to optimize domain-specific tasks. The system is trained on a large, high-quality corpus and is known for its high performance-to-parameter ratio. In contrast, DeepSeek AI's more recent model, DeepSeek R1, is intended to provide robust reasoning and classification capabilities at a lower cost.

It is particularly well-suited for code interpretation tasks since it incorporates reinforcement learning techniques, especially for chain-of-thought (CoT) creation, and is pre-trained on both code and normal language datasets. Although all three models use attention-based processes, there are substantial differences in their access regulations, training data composition, and fine-tuning assistance, which ultimately affects how well they perform in the field of smart contract vulnerability detection.

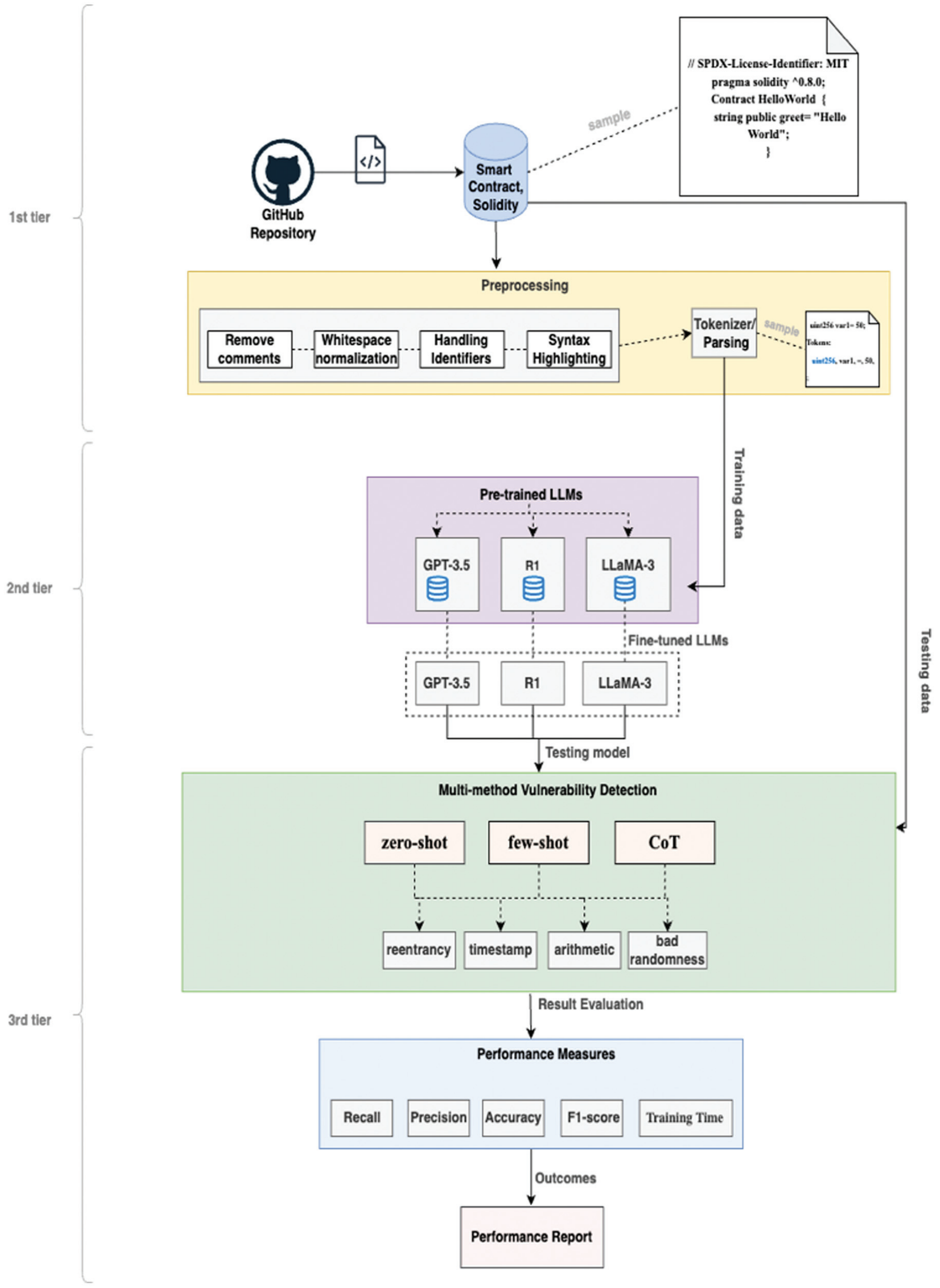
B. DATASET

We utilize a curated open dataset of Solidity smart contracts named Smartcontract-benchmark, which contains 245 smart contracts categorized into four types: reentrancy (81), arithmetic (65), time manipulation (60), and bad randomness (10). The dataset covers various use cases, including token contracts, crowdfunding contracts, decentralized exchange contracts, and wallet contracts. The dataset includes both old and modern Solidity versions, providing a diverse set of code samples for training and evaluating LLMs.

However, the vulnerability distribution in the dataset is unbalanced due to the small number of contracts representing bad randomness vulnerabilities. This is due to common programming mistakes or misunderstood language features leading to reentry and arithmetic issues. Despite this, public datasets rarely contain smart contracts with weak randomness due to developers' awareness of the associated risks.

During model training, data augmentation techniques are used to prevent bias toward more frequent categories and neglect of rare ones. This allows the model to be trained with a more uniform distribution while respecting its original distribution during testing. These variations allow the dataset to better reflect the diversity of Solidity smart contract codes, especially when preparing the model for zero-shot and few-shot evaluation scenarios.

A double-check is conducted to ensure no duplicate contracts or syntactically invalid contracts were included, and preprocessing was performed to normalize contracts. Contracts were tokenized


```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;
Contract HelloWorld {
string public greet= "Hello
World";
}
```

```
uint256 var1= 50;
Tokens:
uint256, var1, =, 50,
;
```

Fig. 1. Proposed model architecture.

using language model-compatible tokenizers, ensuring compatibility with transformer-based architectures such as GPT3.5-turbo, R1, and CodeLlama.

After augmentation the dataset is divided into three distinct subsets, as shown in Table III, with a training set (70%), a validation set (15%), and a test set (15%). The Solidity code in the dataset ranges from simple single-function contracts to large, feature-rich decentralized applications, allowing the model to detect vulnerabilities across a variety of smart contract designs.

C. PROMPTS DESIGN

To evaluate LLMs' effectiveness in detecting vulnerabilities, three different reasoning approaches were employed during inference: zero-shot, few-shot, and CoT shown in Fig. 2. Using different levels of guidance, each approach was designed to simulate smart contract auditor behavior. However, zero-shot prompting involves presenting the model with simple instructions and raw contract code, without prior examples. The prompt was phrased as follows: "You are a Solidity smart contract auditor. Analyze the following contract and identify if there are any vulnerabilities. Clearly

Table III. Number of contracts in each set over variability labels

Dataset	Reentrancy	Arithmetic	Time manipulation	Bad randomness	Total
Training	57	45	42	27	171
Testing	12	10	9	6	37
Validation	12	10	9	6	37
Total	81	65	60	39	245

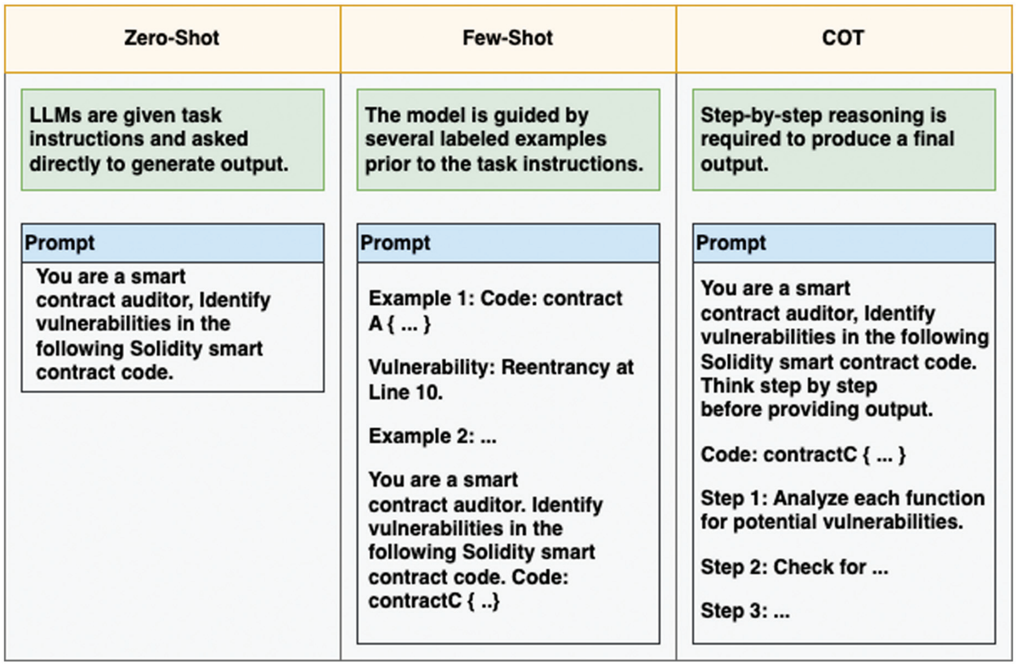

Zero-Shot
LLMs are given task instructions and asked directly to generate output.
Prompt
You are a smart contract auditor, Identify vulnerabilities in the following Solidity smart contract code.
Few-Shot
The model is guided by several labeled examples prior to the task instructions.
Prompt
Example 1: Code: contract A { ... }
Vulnerability: Reentrancy at Line 10.
Example 2: ...
You are a smart contract auditor. Identify vulnerabilities in the following Solidity smart contract code. Code: contractC { ..}
COT
Step-by-step reasoning is required to produce a final output.
Prompt
You are a smart contract auditor, Identify vulnerabilities in the following Solidity smart contract code. Think step by step before providing output.
Code: contractC { ... }
Step 1: Analyze each function for potential vulnerabilities.
Step 2: Check for ...
Step 3: ...

Fig. 2. Brief description and example of zero-shot, few-shot and CoT prompts.

mention the type of vulnerability if found (e.g., reentrancy, arithmetic, time manipulation, bad randomness). If the contract is safe, respond with 'no vulnerability detected'."

In few-shot prompting, it was extended with five examples showing smart contracts and it is existed vulnerability type. As a result of these examples, the model understood the expected output format and reasoning style prior to presenting the actual contract to be evaluated. CoT prompting encourages step-by-step reasoning. Using CoT, the model was instructed to evaluate function logic, and check for specific vulnerability patterns.

The prompt was phrased as follows: "You are a Solidity smart contract auditor. Follow these steps carefully to perform a full vulnerability assessment. (1) List and explain all state/storage variables and their roles. (2) Analyze each function and describe its purpose and execution logic. (3) For each function, identify potential vulnerability types such as bad randomness (e.g., using block.timestamp or block.number), time manipulation (e.g., time-based conditions), reentrancy patterns (e.g., external calls before state updates), arithmetic errors (e.g., overflows/underflows, unchecked math). (4) Highlight the exact lines or patterns that raise concerns. (5) Clearly state if the contract is vulnerable and mention the type. (6) Provide a vulnerability type. Begin your step-by-step reasoning below:"

D. DATA FORMAT

As part of the data preparation process, each model was provided with structured inputs that adhered to the expected formatting protocol. This ensured consistency across model-specific interfaces while also maintaining semantic integrity.

With GT-3.5-Turbo, OpenAI's ChatML conversational format was used. The prompts included a system message describing the role of the model (e.g., "You are an auditor of Solidity smart contracts"), a user message containing the contract code, and an assistant placeholder for generating vulnerability classifications. With zero-shot and chain-of-thought prompting, this format enabled contextual understanding and adherence to instructions.

In DeepSeek-R1, each instance paired a cleaned Solidity contract with a vulnerability label in JSON or CSV formats, resulting in a simplified instruction-response structure. To guide model output, CoT instructions and a "Final Answer" tag were appended, facilitating step-by-step reasoning. This improved interpretability and classification focus.

As part of the LLaMA 3 training architecture, a dialogue-like format was used to create examples that resemble multi-turn instructions. Each entry included tokens distinguishing the role of the system, user, and assistant, allowing the system to follow instructions. A single vulnerability label was expected as a response to the prompts. This ensured consistency across all modes of prompting, including zero shots, few shots, and CoT prompts.

E. EVALUATION METRICS

The evaluations focus on detecting vulnerabilities in Solidity smart contracts and assessing the robustness of LLMs through real-world simulations and performance metrics. However, it examines whether LLMs are effective at detecting vulnerable codes by comparing their results. During the evaluation, a simulation of a real-world scenario is conducted to assess the process. The accuracy and effectiveness of the model are measured by standard performance metrics such as precision, recall, F1-score, accuracy, and training time. Also, a variety of prompting strategies, such as zero-shot and few-shot approaches [31–33], are tested.

IV. EXPERIMENTS SETTING

The study aimed to evaluate models under three reasoning conditions: zero-shot, few-shot, and CoT to detect vulnerabilities in Solidity smart contracts. Local and cloud-based computing resources were used, including an Apple M2 processor, RAM, storage, and the macOS Ventura 13.5 operating system. Google Colab Pro+ was used for scalable parallel training and fine-tuning. Python-based libraries and deep learning frameworks were used to implement transformer models and evaluate LLMs. PyTorch, Hugging Face Transformers, Scikit-learn, Pandas, NumPy, tqdm, Matplotlib, and Google Colab were used for training and evaluation.

The SmartContract-Benchmark dataset was used to prepare LLMs for detecting vulnerabilities in smart contracts. The dataset involved data cleaning, label filtering, data enhancement, formatting, tokenization, splitting, and evaluation. Three reasoning approaches were employed: zero-shot, few-shot, and CoT. Zero-shot prompting presented simple instructions and raw contract code. However, supervised learning was used for training hyperparameters that are illustrated in Table IV below. The fine-tuning technique adapted the model's general knowledge to specific application contexts, enhancing its performance on specific tasks.

A. LLM EXPERIMENTS

The GPT-3.5-turbo model was evaluated and compared with three different prompting strategies. The model uses the ChatML format to define interactions between roles in a conversational setting, ensuring it interprets each part of the conversation correctly and generates responses aligned with its intended behavior. Tokenization is required as part of the preprocessing step, and GPT-3.5-turbo does not require manual tokenization; it is handled automatically via the API. Furthermore, fine-tuning of the GPT-3.5-turbo model was conducted using OpenAI's managed fine-tuning API. Training data were formatted in ChatML with structured system, user, and assistant roles, and each example followed a prompt-response structure. After the preprocessing step, training examples were converted to a JSONL format compatible with OpenAI's API. Each line contained a structured message list for dialogue-style interaction.

While DeepSeek-R1 is a distilled and highly efficient version of DeepSeek-V3-Base, which includes 671 billion total parameters, 37 billion of which are active during inference. Reinforcement learning (RL) is used to generate a CoT reasoning path before delivering its final output, enhancing logic consistency and interpretability. Group relative policy optimization (GRPO) is an important component of its learning strategy, allowing the model to compare its current responses to previous attempts and update its behavior only when it demonstrates improvement. The dataset format included code field, label field, and a prompt-style instruction. A CSV and JSON version of the dataset were provided for training and evaluation. The model started acting as an auditor for smart contracts and analyzed the Solidity code step-by-step. An end-of-sequence (EOS) token was affixed to each sample to maintain consistency in training and inference. The fine-tuning setup uses the unsloth library, which supports low-rank adaptation (LoRA) to tune memory usage. Inference is accelerated by FastLanguageModel.for_inference() without affecting the core behavior of the model. A set of hyperparameters was used to define the training strategy, with a device batch size of two, a gradient accumulation factor of four, and a maximum of 60 steps. Further, LLaMA 3 is a collection of foundational autoregressive language models developed by Meta AI to perform a wide range of natural language processing tasks. The LLaMA 3–8B variant was used in this study, which improves performance, reasoning, and instruction alignment over its predecessors.

Table IV. Training hyperparameters for each model

Hyperparameter	GPT	DeepSeek	LLaMA
Base model	gpt-3.5-turbo-0125	DeepSeek-R1-Distill-Qwen-14B	llama-3-8b
Epochs	3	4	3
Batch size	1	2	2
Learning rate	2	2e-4	3e-5
Tokenizer	Byte pair encoding	AutoTokenizer	AutoTokenizer

V. RESULTS AND DISCUSSION

The study analyzed the performance of various LLM models in identifying vulnerabilities. GPT3.5-turbo fine-tuning was completed in approximately 20 minutes that achieved the highest overall accuracy of 94.59%, while CoT reasoning achieved 86.49%. However, precision and F1-scores remained low across all prompting approaches. Some types of vulnerabilities are more reliable than others, such as arithmetic and reentrancy. The training process was relatively short, demonstrating the feasibility of adapting general-purpose LLMs to specialized tasks without extensive computational resources. DeepSeek-R1 model, an RL system that uses CoT reasoning. It was fine-tuned in 2623.58 seconds, which is approximately 43.7 minutes and achieved an accuracy of 78.95% and a precision of 0.37%. However, it did not conform easily to strict classification metrics due to its outputs.

The LLaMA 3 model showed distinct performance characteristics, with zero-shot setting achieving excellent results with 92% accuracy, precision, recall, and F1 scores. The model was capable of avoiding false positives and false negatives due to its high balance between precision and recall. CoT reasoning strategy produced highly competitive results, with 96% accuracy, 82% F1, 88% precision, and 86% recall. The fine-tuning process for LlaMA 3 model was completed in 16.15 minutes, demonstrating the value of CoT analyses in ensuring transparency in security tasks.

In order to better understand Chain-of-Thought (CoT) prompting, we examined select test cases in which CoT produced accurate

Table V. Accuracy and training time of LLMs

LLMs	Approach	Accuracy	Training time/ min
GPT3.5-turbo	Zero-shot	94.59%	20
	Few-shot	94.59%	
	CoT	86.49%	
DeepSeek-R1	CoT	78.95%	43.7
LlaMA-3	Zero-shot	92%	16.15
	Few-shot	58%, 100% partial	
	CoT	96%	

predictions, but zero-shot and few-shot approaches failed. As an example, CoT prompting enabled a model to explicitly reason through the execution path when a conditional withdrawal pattern was present, recognizing the possibility of reentrancy after the external call in a contract with a conditional withdrawal pattern. The zero-shot prompt, on the other hand, identified balance updates without understanding the execution process. CoT enhances interpretability and logical flow in reasoning-heavy contexts, particularly when vulnerabilities are detected through simulation of contract behavior. However, the study demonstrates the effectiveness of LLMs in detecting vulnerabilities in Solidity smart contracts. Also, Table V illustrates results that highlight the potential of LLMs to handle domain-specific tasks without supervision, bridging the gap between machine learning-based code analysis and real-world smart contract auditing.

VI. COMPARISON OF LLMs

The variation in performance among models can be attributed to differences in prompt alignment, pre-training datasets, and architecture. Since GPT-3.5-Turbo is optimized for instruction following and has a wider coverage of pre-training data, it consistently performs well in zero-shot scenarios. While LLaMA-3 performed exceptionally well in CoT situations due to its flexibility in reasoning-intensive tasks and its refined alignment procedures. On the other hand, DeepSeek R1 demonstrates competitive results because of its design emphasis on multilingual problems and domain-agnostic reasoning. These variations imply that the models' capacity to reason, generalize, and manage complex code semantics in smart contract analysis is directly impacted by architectural choices.

However, Table VI provides a comparison of LLMs for detecting vulnerable Solidity smart contract. According to the results of this study, GPT-3.5-Turbo offers moderate accuracy with high flexibility and is practical and accessible. Structured, accurate classification with minimal preprocessing is provided by DeepSeek-R1, which combines instruction tuning with CoT reasoning. LLaMA-3 provides a cost-effective, open, fine-tuned, and high-performance alternative.

VII. CONCLUSION

To detect vulnerabilities in Solidity smart contracts, this research evaluated three prominent LLMs: GPT-3.5-Turbo, LLaMA-3 (fine-tuned with Unsloth in 4-bit), and DeepSeek R1. Three prompting paradigms were evaluated: zero-shot, few-shot, and CoT. The analysis was performed using the SmartContract-Benchmark dataset, a collection of real-world contracts labeled with vulnerabilities that include reentrancy, arithmetic, time manipulation, and bad randomness.

Throughout the experiments, each model demonstrated its own strengths. LLaMA-3 was highly accurate in identifying specific vulnerability patterns, especially when trained on augmented and cleaned datasets. GPT-3.5-Turbo performed well in generalization tasks with minimal prompt tuning, while DeepSeek R1 performed exceptionally well in reasoning-intensive tasks. All models showed enhanced interpretability and detection accuracy with CoT prompting, confirming the value of step-by-step reasoning.

According to these results, LLMs were viable and promising tools for analyzing smart contracts. Even though they were not perfect, they could already serve as capable assistants or initial filters for auditing systems. As demonstrated by CoT and few-shot strategies, prompt engineering could lead to significant performance gains without requiring massive retraining or custom architectures. BC security tools could be built on this foundation to provide scalable and intelligent capabilities.

A. LIMITATIONS AND FUTURE WORK

Despite promising results across all evaluated models and reasoning strategies, the dataset used has certain limitations. Approximately 245 smart contracts were included in the dataset, but certain classes—such as bad randomness—had significantly fewer examples (39 contracts). Due to this class imbalance and small dataset size, the findings may not be generalizable in real-world scenarios with more diverse contract structures and less frequent vulnerability types.

Yet, while the dataset covers a variety of vulnerability types, the relatively small number of test samples (n = 37) limits the statistical power of the results. The sample size is not sufficient to accurately capture the complexity and variability of real-world smart contracts.

Future research should expand the test set to ensure statistical significance and broader generalization. The model's robustness can also be further validated by including contracts from various sources and applying advanced data augmentation techniques.

Table VI. Comparison of the three models in detecting solidity smart contracts vulnerabilities

Aspect	GPT-3.5-Turbo	DeepSeek-R1	LLaMA 3
Description	API-based, instruction-tuned	Distilled, instruction-tuned	Unsloth 4-bit variant of the LLM (fine-tuned)
Prompting strategies	Zero-shot, few-shot, CoT	CoT	Zero-shot, few-shot, CoT
Source code	Requires OpenAI API access	Open source	Open source
Strengths	Accessible and flexible	Clear and interpretable outputs	Generalized and cost effective
Limitations	Prompt-sensitive outputs, limited control	Not broadly tested	Requires careful prompt design

ACKNOWLEDGMENTS

The authors extend their appreciation to the Deanship of Scientific Research at Northern Border University, Arar, KSA, for funding this research work through the project number "NBU-FFR-2025-1260-10."

CONFLICT OF INTEREST STATEMENT

There is no conflict of interest regarding the publication of this paper

REFERENCES

[1] A. Bouichou, S. Mezroui, and A. E. Oualkadi, "An overview of Ethereum and Solidity vulnerabilities," *Proc. 2020 Int. Symp. Adv. Electr. Commun. Technol. (ISAECT)*, November 2020, pp. 1–7.

[2] X. Zhou, S. Cao, X. Sun, and D. Lo, "Large language model for vulnerability detection and repair: literature review and the road ahead," *arXiv preprint* arXiv:2404.02525, October 2024.

[3] Y. Sun et al., "When GPT meets program analysis: towards intelligent detection of smart contract logic vulnerabilities in GPTScan," *CoRR*, January 2023. Available: https://openreview.net/forum?id=vET8N17YLH (accessed May 6, 2025).

[4] C. Chen et al., "When ChatGPT meets smart contract vulnerability detection: how far are we?," *arXiv preprint* arXiv:2309.05520, September 2023. Available: http://arxiv.org/abs/2309.05520 (accessed March 7, 2024).

[5] E. A. Napoli, F. Barbàra, V. Gatteschi, and C. Schifanella, "Leveraging large language models for automatic smart contract generation," in *Proc. 2024 IEEE 48th Annu. Comput., Softw., Appl. Conf. (COMPSAC)*, July 2024, pp. 701–710.

[6] S. Chatterjee and B. Ramamurthy, "Efficacy of various large language models in generating smart contracts," in *Advances in Information and Communication*, K. Arai, Ed. Cham, Switzerland: Springer Nature, 2025, pp. 482–500.

[7] S. S. Kushwaha, S. Joshi, D. Singh, M. Kaur, and H.-N. Lee, "Ethereum smart contract analysis tools: a systematic review," *IEEE Access*, vol. 10, pp. 57037–57062, 2022.

[8] X. Li, J. Liu, X. Chen, and Q. Zhang, "A symbolic execution-based approach for smart contract vulnerability detection," *IEEE Conference Publication*, 2024. Available: https://ieeexplore.ieee.org/abstract/document/10407615 (accessed November 13, 2024).

[9] C. S. Yashavant, S. Kumar, and A. Karkare, "ScrawlD: a dataset of real world Ethereum smart contracts labelled with vulnerabilities," *arXiv preprint* arXiv:2202.11409, February 2022. Available: http://arxiv.org/abs/2202.11409 (accessed March 12, 2024).

[10] P. Qian, Z. Liu, Q. He, B. Huang, D. Tian, and X. Wang, "Smart contract vulnerability detection technique: a survey," *J. Softw.*, vol. 33, no. 8, pp. 3059–3085, 2022.

[11] 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 *Proc. 36th IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*, November 2021, pp. 227–239.

[12] N. Songsom, W. Werapun, J. Suaboot, and N. Rattanavipanon, "The SWC-based security analysis tool for smart contract vulnerability detection," in *Proc. 2022 6th Int. Conf. Inf. Technol. (InCIT)*, November 2022, pp. 74–77.

[13] N. Ashizawa, N. Yanai, J. P. Cruz, and S. Okamura, "Eth2Vec: learning contract-wide code representations for vulnerability detection on Ethereum smart contracts," *arXiv preprint* arXiv:2101.02377, January 2021.

[14] C. Liu, Z. Sang, L. Duan, W. Ni, W. Wang, and C. Li, "TLSCG: transfer learning-based efficient anomalous smart contract generation to empower unknown vulnerability detection," in *Proc. 2024 IEEE Int. Conf. Web Services (ICWS)*, July 2024, pp. 468–478.

[15] Y. Boxin, "Research on dynamic detection of vulnerabilities in smart contracts based on machine learning," in *Proc. 2024 IEEE 3rd Int. Conf. Electr. Eng., Big Data Algorithms (EEBDA)*, February 2024, pp. 219–223.

[16] D. Han, Q. Li, L. Zhang, and T. Xu, "A smart contract vulnerability detection model based on graph neural networks," in *Proc. 2022 4th Int. Conf. Front. Technol. Inf. Comput. (ICFTIC)*, December 2022, pp. 834–837.

[17] X. Zhang, J. Li, and X. Wang, "Smart contract vulnerability detection method based on Bi-LSTM neural network," in *Proc. 2022 IEEE Int. Conf. Adv. Electr. Eng. Comput. Appl. (AEECA)*, August 2022, pp. 38–41.

[18] Z. Yang, C. Zhou, Q. Xiang, W. Huang, and X. Wang, "Smart contract vulnerability detection based on TextCNN and attention mechanism," *IEEE Conference Publication*, 2024. Available: https://ieeexplore.ieee.org/abstract/document/10483421 (accessed November 7, 2024).

[19] A. Koubaa, "GPT-4 vs. GPT-3.5: a concise showdown," *Preprints*, 2023030422, March 2023.

[20] S. Hu, T. Huang, F. İlhan, S. F. Tekin, and L. Liu, "Large language model-powered smart contract vulnerability detection: new perspectives," *arXiv preprint* arXiv:2310.01152, October 2023. Available: http://arxiv.org/abs/2310.01152 (accessed March 7, 2024).

[21] B. Gao, Q. Wei, Y. Liu, and R. S. M. Goh, "Unveiling the potential of ChatGPT in detecting machine unauditable bugs in smart contracts: a preliminary evaluation and categorization," *IEEE Conference Publication*, 2024. Available: https://ieeexplore.ieee.org/abstract/document/10605444 (accessed November 14, 2024).

[22] B. Boi, C. Esposito, and S. Lee, "VulnHunt-GPT: a smart contract vulnerabilities detector based on OpenAI ChatGPT," in *Proc. 39th ACM/SIGAPP Symp. Appl. Comput.*, Avila, Spain: ACM, April 2024, pp. 1517–1524.

[23] Y. Du and X. Tang, "Evaluation of ChatGPT's smart contract auditing capabilities based on chain of thought," *arXiv preprint* arXiv:2402.12023, February 2024.

[24] C. Wang and M. Kantarcioglu, "A review of DeepSeek models' key innovative techniques," *arXiv preprint* arXiv:2503.11486, March 2025.

[25] R. Karanjai, S. Blackshear, L. Xu, and W. Shi, "A multi-agent framework for automated vulnerability detection and repair in Solidity and Move smart contracts," *arXiv preprint* arXiv:2502.18515, February 2025.

[26] B. Rozière et al., "Code Llama: open foundation models for code," *arXiv preprint* arXiv:2308.12950, January 2024.

[27] M. T. Alam, R. Halder, and A. Maiti, "Detection made easy: potentials of large language models for Solidity vulnerabilities," *arXiv preprint* arXiv:2409.10574, September 2024.

[28] J. Zhang et al., "An empirical study of automated vulnerability localization with large language models," *arXiv preprint* arXiv:2404.00287, March 2024.

[29] Z. Xiao, Q. Wang, H. Pearce, and S. Chen, "Logic meets magic: LLMs cracking smart contract vulnerabilities," *arXiv preprint* arXiv:2501.07058, January 2025.

[30] Z. A. Nazi, M. R. Hossain, and F. A. Mamun, "Evaluation of open and closed-source LLMs for low-resource language with zero-shot, few-shot, and chain-of-thought prompting," *Nat. Lang. Process. J.*, vol. 10, p. 100124, 2025.

[31] R. Alhejaili, A. Alsaeedi, and W. M. S. Yafooz, “Detecting hate speech in Arabic tweets during COVID-19 using machine learning approaches,” In *Proceedings of Third Doctoral Symposium on Computational Intelligence*, A. Khanna, D. Gupta, V. Kansal, G. Fortino, and A. E. Hassanien, Eds. Singapore: Springer Nature, 2023, pp. 467–475.

[32] R. A. M. Alsaidi, W. M. S. Yafooz, H. Alolofi, G. A.-M. Taufiq-Hail, A.-H. M. Emara, and A. Abdel-Wahab, “Ransomware detection using machine and deep learning approaches,” *ProQuest*, 2025. Available: https://www.proquest.com/openview/421f66c9054aaffb5cb94aabdc786fdb/1?pq-origsite=gscholar&cbl=5444811 (accessed July 4, 2025).

[33] I. Chamieh, T. Zesch, and K. Giebermann, “LLMs in short answer scoring: limitations and promise of zero-shot and few-shot approaches,” *ACL Anthology*, 2024. Available: https://aclanthology.org/2024.bea-1.25/ (accessed May 8, 2025).