Large Language Model-Powered Smart Contract Vulnerability Detection: New Perspectives

Sihao Hu, Tiansheng Huang, Fatih İlhan, Selim Furkan Tekin, Ling Liu

School of Computer Science
Georgia Institute of Technology
Atlanta, GA 30332, United States
{sihaohu, thuang, filhan, stekin6, ling.liu}@gatech.edu

Abstract—This paper provides a systematic analysis of the opportunities, challenges, and potential solutions of harnessing Large Language Models (LLMs) such as GPT-4 to dig out vulnerabilities within smart contracts based on our ongoing research. For the task of smart contract vulnerability detection, achieving practical usability hinges on identifying as many true vulnerabilities as possible while minimizing the number of false positives. Nonetheless, our empirical study reveals contradictory yet interesting findings: generating more answers with higher randomness largely boosts the likelihood of producing a correct answer but inevitably leads to a higher number of false positives. To mitigate this tension, we propose an adversarial framework dubbed GPTLENS that breaks the conventional one-stage detection into two synergistic stages – generation and discrimination, for progressive detection and refinement, wherein the LLM plays dual roles, *i.e.*, AUDITOR and CRITIC, respectively. The goal of AUDITOR is to yield a broad spectrum of vulnerabilities with the hope of encompassing the correct answer, whereas the goal of CRITIC that evaluates the validity of identified vulnerabilities is to minimize the number of false positives. Experimental results and illustrative examples demonstrate that AUDITOR and CRITIC work together harmoniously to yield pronounced improvements over the conventional one-stage detection. GPTLENS is intuitive, strategic, and entirely LLM-driven without relying on specialist expertise in smart contracts, showcasing its methodical generality and potential to detect a broad spectrum of vulnerabilities. Our code is available at: **https://github.com/git-disl/GPTLens**.

Index Terms—Large language model, GPT, smart contract, vulnerability detection

I. INTRODUCTION

Smart contracts, commonly associated with cryptocurrency transactions on blockchains, were racked with financial losses up to billions of dollars due to vulnerability exploitation [59]. Due to the immutable nature once smart contracts are deployed, auditing acts an essential role in their development. Recently, generative Large Language Models [5], [26], [58] (LLMs) are rapidly emerging and reshaping the domain of software engineering [15], facilitating tasks of code generation [9], code understanding [46], and code repair [28]. Leveraging the capabilities of LLMs to empower smart contract auditing presents a promising application opportunity. In this paper, we envision the development of LLM-powered smart contract vulnerability detection techniques from a new perspective, in tandem with a systematic analysis of the opportunities and challenges involved in this nascent research topic.

Compared to the existing representative analysis tools [3], [4], [11], [19], [24], [44] developed in the past years, LLM-powered detection features some unparalleled advantages:

(1) *Generality*: Existing tools like Slither [11] require expert knowledge to design fixed-pattern detectors based on control-flow or data-flow, restricting them to specific types of vulnerabilities [48]. In contrast, LLMs can emulate human linguistic understanding and reasoning and describe any type of vulnerability using natural language, allowing them to potentially detect a wider range of vulnerabilities, including those that are unknown or uncategorized a priori.

(2) *Interpretability*: Generative LLMs can be utilized not only to detect vulnerabilities but also to offer intermediate reasoning about the detected vulnerabilities by following the chain-of-thought [47]. For programming and software engineering tasks, LLMs can provide insights into code understanding [46] and even suggest code repair solutions [28]. Such capabilities, if exploited intelligently, hold the potential to grant a heightened level of transparency and trustworthiness to the vulnerability detection process.

Nevertheless, certain limitations hinder LLMs from being exploited to their full potential:

(1) LLMs can produce a large number of false positives [10], resulting in a low precision and necessitate exhausting manual verification efforts. These false positive cases, categorized as factual errors or potential vulnerabilities, suggesting that they should be differentiated from the true vulnerabilities in terms of metrics beyond just correctness.

(2) LLMs, if used in a naive manner, tend to fail to uncover all vulnerabilities within the smart contract, leading to false negatives. These undetected vulnerabilities can be categorized into two primary groups: First, hard cases that exceed the "cognitive ability" of current LLMs; Second, vulnerabilities that are detectable but were missed because of the randomness of the generation. For the latter, our empirical study shows that instead of generating deterministic answers in one-shot, generating multiple answers with higher randomness (diversity) can largely boost the likelihood of the true answer being generated. Nevertheless, this strategy presents a Catch-22 dilemma [14] as it inevitably introduces more false positives, *i.e.*, the goal of detecting more true vulnerabilities is misaligned with the goal of reducing false positives.

To mitigate this tension, we propose GPTLENS, a frame-

work that separates the conventional one-stage detection into two adversarial yet synergistic stages: *generation* followed by *discrimination*.

The primary goal of the generation stage is to enhance the likelihood of true vulnerabilities being identified (generated). With this goal in mind, we ask an LLM to play the role of multiple auditor agents, and each auditor generates answers (vulnerability and the reasoning) with high randomness, even though it could lead to a plethora of incorrect answers. In contrast, the goal of the discrimination stage is to discriminate between true and false answers generated in the generation stage. To realize this, we prompt the LLM to play the role of a critic agent, which evaluates each identified vulnerability on a set of criteria, such as correctness, severity, and profitability, and assigns corresponding scores. Subsequently, GPTLENS ranks the answers by these scores to select the top-k results.

The primary advantage of GPTLENS is that it resolves the Catch-22 dilemma present in one-stage detection, i.e., the conflicting goals between increasing the probability of generating the correct answer and reducing the number of false positives. Furthermore, GPTLENS is a *pure* LLM-driven framework without resorting to any expert knowledge during the end-to-end vulnerability detection process.

We conduct preliminary experiments on 13 real-world smart contracts, all of which were reported to contain vulnerabilities in the Common Vulnerabilities and Exposures (CVEs) database [45]. Experiments indicate that compared to the conventional one-stage detection which identifies true vulnerabilities in 38.5% of contracts with the top-1 output, GPTLENS succeeds in 76.9% of contracts. When comparing at the trial level, the accuracy for top-1 results rises from 33.3% to 59.0%. This enhancement is exciting as GPTLENS is simple and does not rely on any intricate design, suggesting its potential for broader application scenarios.

In summary, this paper makes three original contributions:

- We provide a *systematic* analysis of the advantages (opportunities) and challenges of LLM-powered smart contract vulnerability detection techniques.
- We introduce an *innovative* framework, GPTLENS, constituted by two adversarial yet synergistic stages wherein the LLM takes on the roles of the auditor and critic agents respectively.
- GPTLENS is *simple*, *effective* and *purely* LLM-driven, eliminating the need for specialist expertise and showing the potential for generalization across a range of vulnerability types.

II. LLM-POWERED VULNERABILITY DETECTION

A. Standard Detection Paradigms

There are three standard prompting paradigms for vulnerability detection, *i.e.*, binary prompting, multi-class prompting and open-ended prompting.

Existing works primarily follow the binary prompting paradigm [10], [40]. In this paradigm, LLMs are prompted with the smart contract code and a specific vulnerability

> **Binary prompt:** You are a smart contract auditor. Review the following smart contract code in detail. Is the contract vulnerable to {vul_type}? Reply with YES or NO only. {contract code}

type [48], such as integer overflow/underflow, re-entrancy, or access control risk, and is expected to produce a binary YES or NO answer. These studies [10], [40] also recommend including the definition or additional information about the vulnerability type to enhance performance. Given n categories of vulnerabilities, the LLM service should be queried n times for each smart contract.

> **Multi-class prompt:** You are a smart contract auditor. Here are {n} vulnerabilities: {vul_type1, vul_type2, ..., vul_type$n$}. Review the following smart contract code in detail. Use 0 or 1 to indicate the presence of specific types of vulnerabilities, such as {vul_type1: 0, vul_type2: 1, ..., vul_type$n$: 0}. {contract code}

An extension of binary prompting is multi-class prompting [8], which requires LLMs to categorize identified vulnerabilities into multiple classes. Both binary and multi-class prompting fall under the category of *close-ended* prompting, as they necessitate that the vulnerability categories be *predefined*. Nevertheless, there always exist vulnerabilities that are either unknown or not categorized: Zhang et al. [57] found that 80% of exploitable bugs remain undetected by existing analysis tools.

> **Open-ended prompt:** You are a smart contract auditor. Review the following smart contract code in detail and identify vulnerabilities within it. {contract code}

In this paper, we propose a new prompting paradigm dubbed *open-ended* prompting, which prompts LLMs to identify any potential vulnerabilities they think might be, and describe them in natural language without being constrained by predefined vulnerability names, theoretically enabling LLMs to recognize a broader range of vulnerability types.

B. Advantages of LLM-powered Detection

Interpretability: Beyond merely identifying vulnerabilities, we can ask LLMs to produce explanations for code, generate intermediate reasoning for vulnerability detection, generate examples of how to exploit identified vulnerabilities, and suggest code repair solutions. Such interpretability offers a new degree of transparency and trustworthiness, as we can gain insight into the step-by-step thought process [47] of LLMs. A case study in Listing 3 presents the explanations provided by GPT-4 for identified vulnerabilities.

Generality: Traditional smart contract auditing tools [4], [11], [19], [24], [44] have difficulty detecting unknown or

uncategorized vulnerabilities since detectors are predesigned by human experts for fixed patterns of specific vulnerability types.

Existing AI-powered detection methods [17], [32], [60] also feature limited generality because they work in a supervised classification manner: vulnerabilities are classified into a fixed set of predefined categories based on known threats, which are used as the ground-truth to train a detection model.

For LLM-powered detection, although close-ended prompting also requires vulnerabilities to be predefined, open-ended prompting breaks this constraint. To retain the characteristic of generality, we adopt open-ended prompting throughout the paper. In our experiments, when prompting GPT-4 [26] with an open-ended prompt, it identifies "condition logic error" for CVE 2018-11411 and "incorrect constructor name" for CVE 2019-15079, which are semantically precise and fall outside of existing popular categorizations [48].

Efficiency: LLM services provide efficient online inference, making LLM-powered methods output results much faster than many traditional methods [8]. However, pre-training an LLM offline is prohibitively expensive in terms of both computational resources and time [43].

C. Limitations of Current LLM-powered Detection

Although LLM-powered detection offers promising advantages and despite the growing hype and claims regarding what LLMs can do, our empirical study has exposed some limitations inherent in current LLMs, which inhibit them from reaching their full potential in practice. Below, we discuss two primary limitations.

```
1 function multiTransfer(address[] _addresses,
     uint[] _amounts) public returns (bool
     success) {
2   require(_addresses.length <= 100 &&
       _addresses.length == _amounts.length);
3   uint totalAmount;
4   for (uint a = 0; a < _amounts.length; a++)
5     totalAmount += _amounts[a];
6   require(totalAmount > 0 && balances[msg.
       sender] >= totalAmount);
7   balances[msg.sender] -= totalAmount;
8   for (uint b = 0; b < _addresses.length; b
       ++) {
9     if (_amounts[b] > 0) {
10        balances[_addresses[b]] +=
            _amounts[b];
11        Transfer(msg.sender, _addresses[b
            ], _amounts[b]);
12    }
13  }
14  return true;
15 }
16 ...
```
Listing 1. Code snippet from the smart contract reported in CVE 2018-13836

1) Large number of false positives: The paramount challenge is that LLMs can produce a large number of false positives (FP), leading to exhausting manual verification efforts. A recent measurement study [10] on project-level vulnerability

detection demonstrates that GPT-4 can only identify 32 out of 73 vulnerabilities on 52 DeFi attacks, but produces 740 false positive cases, leading to an extremely low precision of 4.15% ($Precision = \frac{TP}{TP+FP}$). A similar conclusion can be drawn from the results of Claude [2], which achieves a precision of 4.3%. Another measurement study [8] shows that GPT-3.5 and GPT-4 achieve precisions of 19.7% and 22.6% respectively, in detecting the 9 most common categories of vulnerabilities.

In practice, we observe that false positives can primarily be broken down into two cases:

- *Factual error*: LLMs are insensitive to certain types of syntactic details, such as modifier statements, condition statements, error handling statements (*require, assert, revert*), and event statements, especially when the number of input tokens is huge. For example, in Listing 1, GPT-4 flags the *multiTransfer* function for re-entrancy risk because it believes "the function does not follow the Check-Effects-Interaction pattern, *e.g.*, the state should not be updated before calling external contracts." Nonetheless, this false alarm stems from mistaking *Transfer* as an external function call when it is actually an event statement for data logging without invoking external contracts.

```
1 function setOwner(address _owner) returns (
     bool success) {
2   owner = _owner;
3   return true;
4 }
5 ...
6 function uploadBalances(address[] addresses,
     uint256[] balances) onlyOwner {
7   require(!balancesLocked);
8   require(addresses.length == balances.
       length);
9   uint256 sum;
10  for (uint256 i = 0; i < uint256(addresses.
       length); i++) {
11    sum = safeAdd(sum, safeSub(balances[i],
         balanceOf[addresses[i]]));
12    balanceOf[addresses[i]] = balances[i];
13  }
14  balanceOf[owner] = safeSub(balanceOf[owner
       ], sum);
15 }
16 ...
```
Listing 2. Code snippet from the smart contract reported in CVE 2018-10666

- *Potential vulnerability:* In another case, the identified risk does exist but remains unexploited, possibly because it is neither severe enough nor financially beneficial for attackers. In Listing 2, GPT-4 highlights two vulnerabilities: the "lack of access control" in the *setOwner* function, which allows anyone to call it, and the "arbitrary balance manipulation" in the *uploadBalances* function, enabling the owner to set balances for any addresses arbitrarily, which could inflate the token supply. Despite both of them are correct vulnerabilities, only the former was exploited by the attacker and labeled as a CVE while the latter is considered as a false positive, since the formor is more

severe and profitable. This observation indicates that vulnerability detection should consider not only correctness but also severity and profitability. Vulnerabilities detected in a smart contract should be ranked taking into account all these aspects.

While a recent effort [40] seeks to mitigate the impact of false positives by utilizing sophisticated *rules* for filtering, designing such rules demands expert knowledge and remains effective only for predefined vulnerability types.

2) Large number of false negatives: LLMs fail to detect a large portion of true vulnerabilities, resulting in a low recall ($Recall = \frac{TP}{TP+FN}$). As demonstrated in [10], GPT-4 and Claude-1.3 achieve recalls of 43.8% and 35.6% respectively on 52 DeFi attacks. In our experiments, we observe that false negatives can also be divided into two categories:

- *Hard cases* that are beyond the cognitive capabilities of current LLMs. It is reasonable to assume that certain vulnerabilities surpass the detection abilities of existing LLMs, including intricate logic issues that might elude even human auditors. To detect these hard cases, we expect more powerful LLMs in the future or more complicated designs, which will be discussed in Section V.
- Vulnerabilities that are detectable but *undetected* due to the randomness of generation. As is known, GPT-like LLMs generate text by repeatedly estimating probability distributions for next positions across the vocabulary and sampling token-by-token [1]. During generation, a hyper-parameter t (temperature) controls the sharpness of the distribution [6]. Low randomness leads the LLM to generate more credible results, which outperforms than high randomness when the number of generated samples is small. However, even though high randomness leads to less credible results, it is more likely to generate a correct answer when the times of generation is huge. This observation cannot only be corroborated by our experiments (Section IV), but also by the Codex paper [9]: Figure 1 shows the pass probability of the best result picked out of k samples generated by Codex [9] (a sibling of GPT-3) on a code generation task (HumanEval). When the number of samples reaches 100, a higher temperature (0.8) outperforms a low temperature (0.2) with a huge margin (13 absolute percentage).

Nevertheless, pass@k in Figure 1 is calculated utilizing an oracle (ground-truth knowledge), which is not available in real-world applications and generating more diverse answers inevitably leads to more false positives. Hence, how to identify more correct vulnerabilities without introducing more false positives is a challenge for LLM-powered detection.

III. Two-Stage Adversarial Detection Framework

Goal: Our goal is to present a simple, effective, and entirely LLM-driven methodology to shed light on the design of LLM-powered approaches. For every identified vulnerability, it should indicate the associated function and provide the reasoning.
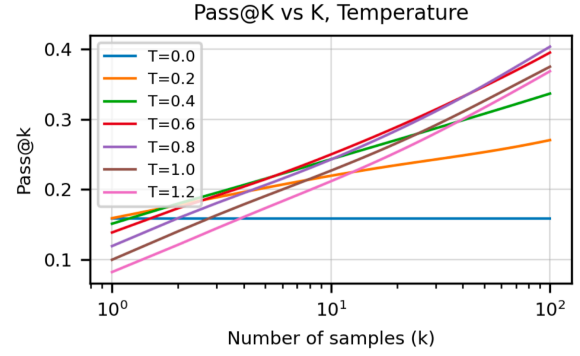

Fig. 1. Pass@k against the number of samples (k) *w.r.t.* various temperatures reported in the Codex paper [9]. Pass@k is the probability of the best result out of k generated samples, where the best sample is picked by an oracle (ground-truth knowledge). Higher the temperature t, higher the randomness of samples. When $t = 0$ the LLM generates deterministic results. higher temperatures are better when the number of samples is large.

Motivations: The design of GPTLENS is inspired by our previous analyses, summarized as follows:

(1) Open-ended prompting has good generalization across a wide range of vulnerabilities, including those unknown or uncategorized ones.
(2) Reducing false positives is crucial for practical applications. False positives should be assessed not only for correctness but also for severity and profitability.
(3) Generating a larger set of diverse samples can raise the likelihood of generating the correct answer, but it inevitably leads to more false positives.

The objective of identifying more correct vulnerabilities is *in conflict* with the goal of reducing false positives in the current one-shot detection paradigm. To mitigate this tension, we break one-shot detection into two adversarial stages, *i.e.*, the generation stage and discrimination/ranking stage. The idea of dividing one-stage into multiple stages is also employed in industrial recommendation systems [7] to optimize respective goals at different stages.

Figure 2 shows the overall framework of GPTLENS, where an LLM plays the roles of two adversarial agents, *i.e.*, the *auditor* and the *critic*, activated by different prompts in respective stages.

In the generation stage, multiple auditors independently audit the smart contract code, generating identified vulnerabilities along with their associated functions and reasoning. The goal of this task is to yield a broad spectrum of answers, with the hope of encompassing the correct one.

In the discrimination stage, the identified vulnerabilities and their associated reasoning are scrutinized, evaluated and ranked by the critic agent, taking into account factors such as correctness, severity and profitability. The goal of this task is to simulate the role of an oracle, *i.e.*, to precisely discern the correct answer and rank it above all other false positives. It is worth noting that the discrimination is not solely based on the identified vulnerability, but heavily leans on reasoning provided by auditors.
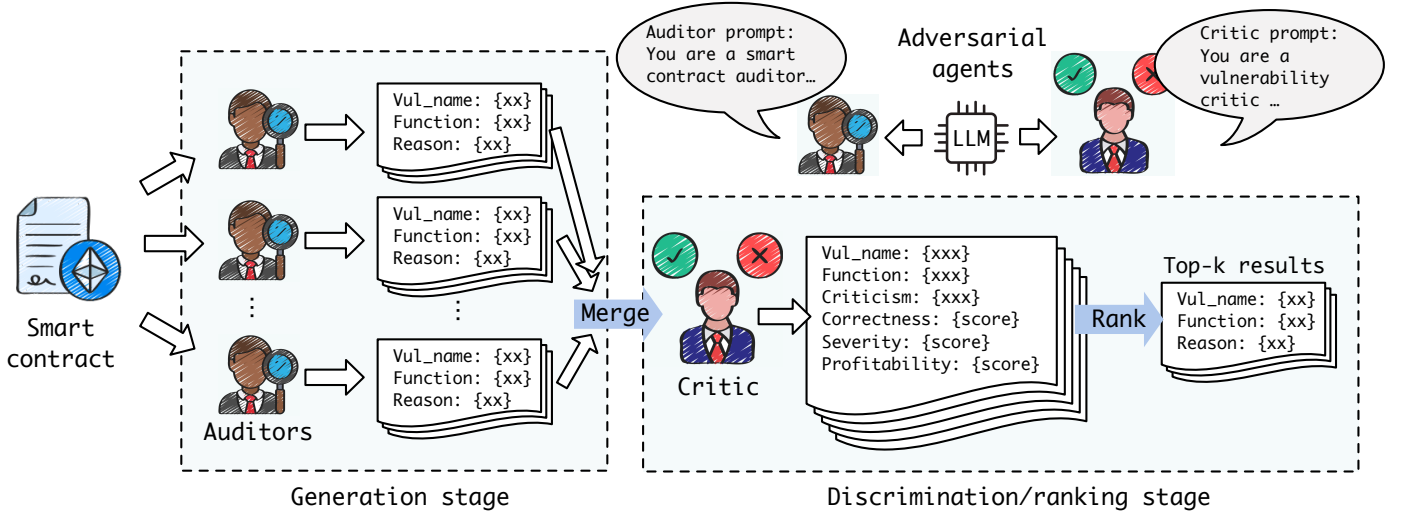
Fig. 2. GPTLENS: an adversarial framework that breaks the conventional one-stage detection into the generation and discrimination stages.

Proof of feasibility: Before diving into the details of GPTLENS, we raise a pertinent question: *How is the critic agent able to discriminate between answers generated by the auditor agents if all the agents are played by the same LLM?*

The answer lies in the difficulty of two tasks: the generation task is more challenging than the discrimination task because the former demands the model to dig out all possible vulnerabilities within an entire smart contract project, while the latter only requires an assessment of recognized vulnerabilities at the function level. Moreover, the success of the adversarial framework has been previously corroborated by GANs [12], wherein a very simple discriminator can be employed to train a sophisticated generator [33].

To provide a clearer picture, in Figure 3 we present the reasoning generated by an auditor along with the criticism from the critic for the code snippet in Listing 2. The critic concludes that the vulnerability identified by the auditor is a factual error.

> **Reasoning by the auditor**: There is a potential *re-entrancy attack* within the *MutiTransfer* function because it does not follow the Checks-Effects-Interactions pattern. The state is not updated before calling external contracts, which could potentially lead to re-entrancy attacks.
>
> **Criticism from the critic**: In the given *multiTransfer* function, there doesn't appear to be any external calls before state changes. The function updates the balances appropriately before emitting a Transfer event. It's worth noting that events like *Transfer* don't actually invoke external contract functions. They simply log data. Therefore, the vulnerability description "Potential for Re-Entrancy Attack" seems *incorrect* based on the provided code.

Fig. 3. Reasoning and criticism for code snippet in Listing 2

> **Auditor prompt:** You are a smart contract auditor, identify and explain severe vulnerabilities in the provided smart contract, ensuring they are real-world exploitable and beneficial to attackers. Include reasoning and corresponding function code as well. Output up to $\{m\}$ most severe vulnerabilities. If no vulnerabilities are detected, output "null".
> {contract code}
> Remember your output should adhere to the following format: {json format}.

Fig. 4. Auditor prompt

A. Generation Stage

The goal of the generation stage is to identify a candidate set of vulnerabilities that includes the correct answer. Therefore, we set up n auditors and initialize them with the auditor prompt presented in Figure 4. Each auditor is responsible for identifying up to m vulnerabilities within the given smart contract code. The auditors' output strictly follows a JSON format for ease of parsing. Temperature t is set based on the hyper-parameters n and k since the optional value for t varies depending on the number of samples. If n is large, t should also be set higher to prevent multiple auditors from generating similar answers.

> **Critic prompt:** As a meticulous and harsh critic, your duty is to scrutinize the function and evaluate the identified vulnerabilities and reasonings with scores in terms of correctness, severity and profitability. Your criticism should include an explanation for your scoring.
> {output of auditors}
> Remember your output should adhere to the following format: {json format}.

Fig. 5. Critic prompt

B. Discrimination/ranking Stage

The role of the critic agent is to emulate an oracle, *i.e.*, to discern the best answer from a multitude of false positives. To ascertain what is the best, we consider three distinct factors: correctness, severity and profitability, because although some false positives may be correct, they might possess a diminished level of severity or may not be profitable for attackers.

Concretely, the critic agent is activated using the critic prompt shown in Figure 5, which directs the critic to evaluate the vulnerability, assign scores based on its reasoning, and provide explanation for these scores. Subsequently, we rank all vulnerabilities descendingly based on these scores and choose the top-k vulnerabilities from the list as the output. Ideally, if the input list of vulnerabilities contains the ground-truth answer, the critic will place it at the forefront.

In our experiments, we employ only one critic agent to ensure that the criticism and scoring remain consistent across various vulnerabilities. Moreover, we set a low temperature value to reduce randomness.

IV. EXPERIMENT

In this section, we validate the previous analyses and the efficacy of GPTLENS via experimental results.

A. Experimental Settings

Dataset: We collected the source code of 13 smart contracts from Etherscan[1], with each containing a reported vulnerability. We sourced the labels and descriptions for these vulnerabilities from the CVE database [45].

Competitors: The experiment involves six competitors under different settings. For methods, A, R, C, O represent Auditor, Random, Critic and Oracle. For parameters, n denotes the number of auditors and m denotes the maximum number of vulnerabilities identified by each auditor. The notation (n=2,m=3) means there are 2 auditors, and each auditor can generate up to 3 vulnerabilities. GPT-4 is adopted as the backend LLM. The descriptions for six competitors are as follows:

- A(n=1,m=1): One auditor identifies up to one vulnerability as the output (*aka* one-stage detection).
- A+R(n=1,m=3): One auditor identifies up to three vulnerabilities and randomly pick one as the output.
- A+C(n=1,m=3): One auditor identifies up to three vulnerabilities, and the critic scores them. The vulnerability with the highest score is selected as the output.
- A+O(n=1,m=3): One auditor identifies up to three vulnerabilities and an oracle is adopted to pick the best answer as the output.
- A+C(n=2,m=3): Two auditors identify up to three vulnerabilities per each and the critic scores them. The vulnerability with highest score is selected as the output.
- A+O(n=2,m=3): Two auditors identify up to three vulnerabilities per each and an oracle is adopted to pick the best answer as the output.

[1]https://etherscan.io

TABLE I
HIT TIMES ON 13 SMART CONTRACTS LOGGED IN CVE DATABASE.

Method	A	A+R	A+C	A+O	A+C	A+O
Parameter	$n1m1$	$n1m3$	$n1m3$	$n1m3$	$n2m3$	$n2m3$
2018-10299	3	1	2	3	3	3
2018-10666	3	1	3	3	3	3
2018-11335	1	1	1	3	1	3
2018-11411	0	2	2	3	2	3
2018-12025	0	0	2	3	3	3
2018-13836	2	0	1	1	0	2
2018-15552	0	0	1	2	2	3
2018-17882	0	0	2	2	3	3
2018-19830	0	1	2	2	3	3
2019-15078	0	0	0	0	0	0
2019-15079	0	0	0	0	0	0
2019-15080	3	1	2	3	3	3
2018-18425	0	0	0	0	0	0
Hit # (CVE)	5	6	**10**	10	9	10
Hit ratio (CVE)	38.5%	46.2%	**76.9%**	76.9%	69.2%	76.9%
Hit # (trail)	13	7	**18**	25	**23**	29
Hit ratio (trail)	33.3%	18.0%	**46.2%**	64.1%	**59.0%**	74.4%

For all the auditors, the temperature t is set to 0.7, while for the critic, t is set to 0 to achieve confident and consistent scoring. It should be noted that the oracle leverages the ground-truth label information, therefore A+O only demonstrates an ideal performance. Due to the constraint on the token number per query, larger n and m are not tested in our experiments.

B. Performance Comparison

Each detection is conducted over three trials. For each trial, up to one vulnerability is selected as the output. A trial is deemed successful only if the function, vulnerability, and reasoning are all in alignment with the CVE report. The number of successful trials is presented in Table I, where Hit # (CVE) is the number of smart contracts for which the method correctly detected vulnerabilities at least once, and Hit # (trial) represents the number of successful trials conducted across 13 smart contracts.

From Table I we can make several observations:

(1) At the trial level, A+R(n=1,m=3) performs worse than A(n=1,m=1), suggesting that generating more answers introduces more false positives. Nonetheless, at the contract level, A+R(n=1,m=3) identifies some CVEs are not detected by A+R(n=1,m=1) like 2018-11411 and 2018-19830, which implies that generating more answers increases the likelihood of generating correct answer.

(2) A more evident observation to support the above argument is that A+O(n=1,m=3) works significantly better than A(n=1,m=1).

(3) A+C(n=1,m=3) outperforms A+R(n=1,m=3), indicating that the critic agent is very *crucial* to discern true vulnerabilities from false positives introduced by generating more answers. A+C(n=1,m=3) also works better than A(n=1,m=1): the Hit ratio (CVE) increases from 38.5% to 76.9%, and the Hit ratio (trail) increases from 33.3% to 46.2%.

(4) Increasing the number of auditors can further improve the performance: compare A+C(n=1,m=3) with A+C(n=2,m=3), the Hit ratio (trail) increases from 46.2% to 59.0%.

C. Case Study

We provide a case study to demonstrate how GPTLENS performs by taking the smart contract code presented in Listing 4 (see Appendix) as the input, with n and m set to 1 and 3 correspondingly. The outputs of both the auditor and critic are presented in Listing 3.

In the generation stage, the auditor identifies three vulnerabilities and their associated reasoning: (1) "Race condition" in the *approve* function (line 101-105); (2) "Race condition" in the *transferFrom* function (line 93-100); (3) "Unexpected behvaior" in the *UBSexToken* function (line 115-119). Although all explanations seem plausible, the first and second vulnerabilities are actually false positives. For the first vulnerability, the LLM overlooks the condition statement in line 102. For the second vulnerability, the LLM disregards the fact that Ethereum transactions are executed sequentially and atomically. As for the third vulnerability, its name differs from "Constructor typo," which is labeled as the ground truth, Nevertheless, the reasoning and identified function are accurate.

In the discrimination stage, the critic uses the vulnerabilities, function code, and the auditor's reasoning as input and outputs criticism and scores with respect to correctness, severity, and profitability. As shown in Listing 3, the critic determines that the reasoning for the first and second vulnerabilities is incorrect, assigning them low scores. Conversely, the critic recognizes that the reasoning and identification of the third vulnerability are accurate, assigning it the highest scores. Consequently, GPTLENS successfully identifies the genuine vulnerability within the smart contract.

As we can see, the entire detection process neither involves manually-defined heuristics and rules for fixed-pattern recognition nor requires a predefined set of vulnerability types, showing that GPTLENS holds the potential to generalize well across a wide range of vulnerabilities.

V. FUTURE DIRECTIONS

To develop a general, practical LLM-powered smart contract vulnerability detection method, several directions can be explored in the future:

Diversity in generation: Increasing the diversity of the generation task helps to enhancing the finding of more true positives. It is benefical to explore innovative prompt engineering strategies and diversity metrics to encourage more generation and auditing diversity while maintaining the same number of generated samples.

Consistency in discrimination: Due to token count constraints, a large number of input vulnerabilities for the discrimination task need to be divided into multiple batches. This division can lead to scoring inconsistencies across different batches, even with a low temperature setting. In-context learning, using few-shot examples, could be explored to teach the

```
{
    "function_name": "approve",
    "vulnerability": "Race Condition",
    "auditor": "This function does not reset
        the allowance before setting the new
        one. This can be exploited by the
        spender by front running the approver,
        allowing them to increase their
        allowance.",
    "critic": "The statement is correct in
        pointing out that this function does
        not inherently reset the allowance.
        However, the function does include a
        check to ensure that if the allowance
        is non-zero, the new value must be
        zero (and vice versa). Therefore, the
        risk is not as serve as stated.",
    "correctness": 3,
    "severity": 0,
    "profitability": 0
}
{
    "function_name": "transferFrom",
    "vulnerability": "Race Condition",
    "auditor": "The _allowance is fetched
        before the transfer is made, making it
        susceptible to a race condition. An
        attacker could call approve() in a
        transaction with a higher gas price,
        allowing them to increase the
        allowance before the transfer is
        executed.",
    "critic": "The statement is incorrect. The
        atomicity and sequential execution of
        Ethereum transactions ensure that
        there's no race condition between
        fetching _allowance and executing the
        token transfer.",
    "correctness": 2,
    "severity": 0,
    "profitability": 0
},
{
    "function_name": "UBSexToken",
    "vulnerability": "Unexpected Behaviour",
    "auditor": "The function name does not
        match the contract name. This
        indicates that this function is
        intended to be the constructor, but it
        is not. This means that anyone can
        call the function at any time and
        reset the totalSupply and owner of the
        token.",
    "critic": "The reasoning is correct. The
        function name does not match the
        contract name, indicating that it is
        intended to be the constructor but it
        is not. This means that anyone can
        call the function at any time and
        reset the totalSupply and owner of the
        token, which is a severe problem.",
    "correctness": 9,
    "severity": 9,
    "profitability": 9,
}
```

Listing 3. A case study on CVE 2018-19830.

LLM for more consistent scoring on novel observations and unseen events.

Reasoning process optimization: A popular direction to enhance the efficacy of LLMs is to design intricate reasoning processes that mimic the thought process of humans, such as chain-of-thoughts [47], tree-of-thoughts [54] and cumulative reasoning [56], which can be adapted for the vulnerability detection task.

Integrating Generative AI agents: Generative AI agents [27], [51] employ LLMs as the core component and perform specific roles in various tasks like software development [30]. In this paper, we design only two synergistic roles for agents. Exploring additional roles for AI agents to achieve sophisticated functionalities, as well as designing how these agents collaboratively interact to solve complex detection tasks, are promising directions.

Enabling external knowledge plug-in: LLMs feature capabilities to use tools or call external APIs to expand their contextual knowledge [34]. It would be intriguing to explore this functionality by allowing the LLM to autonomously determine when and what knowledge is beneficial for generating the correct answer for the vulnerability detection task.

LLM-assisted tools: Instead of serving as an end-to-end solution, LLM can be utilized as a tool to assist developers and auditors throughout the entire software engineering, including code generation [9], [20], code understanding [36], [46], vulnerability detection [42], [49] and code repair [28], [52], to name a few.

VI. RELATED WORK

Various research efforts are dedicated to detecting vulnerabilities in smart contracts.

Traditional methods: Static analysis tools like Securify [44], Vandal [4], Zeus [19], and Slither [11] examine the source code without execution, aiming to detect potential vulnerabilities based on code patterns and structures. In comparison, dynamic analysis tools [13], [18], [50], [55] employ fuzz testing techniques that generates test inputs to identify anomalies during the actual execution of smart contracts, providing insights into runtime vulnerabilities. Symbolic execution tools like Manticore [24] and Mythril [25] investigate vulnerabilities across both bytecode and source code levels by examining all possible execution paths. Additionally, formal verification techniques, such as Verx [29] and VeriSmart [37], validate smart contracts against user-defined specifications, ensuring adherence to desired properties.

DL-powered methods: Deep learning (DL)-based methods like sequence-based models [31], [41], CNN-based methods [39], graph neural networks-based methods [22], [23], [60] are proposed to extract high-level representations to enhance the efficacy of vulnerability detection. Some hybrid methods [21], [35], [53] combines deep-learning techniques with traditional methods. For examples, ESCORT [35] and xFuzz [53] distill the outputs of traditional methods like Slither and Mythril to achieve good generality and inference efficiency. Some works [17], [38] equipped with more advanced

NLP techniques like BERT. A BERT-based approach [16] also demonstrates promising efficacy in Ethereum fraud detection tasks.

Generative LLM-powered methods: Very recent, some studies [8], [10] measure the performance of LLMs on the real-world datasets, suggesting that LLMs face precision-related challenges due to a high occurrence of false positives. GPTScan [40] is introduced in an attempt to mitigate false positives by utilizing rule-based pre-processing and post-confirmation, which requires expert knowledge and extensive engineering efforts. In comparison, GPTLENS is more lightweight and entirely LLM-driven, making it general for a broader range of vulnerabilities.

VII. CONCLUSION

This study provides a systematical analysis of harnessing generative LLMs for smart contract auditing, especially on the challenges of balancing the generation of correct answers against the backdrop of false positives. To address this Catch-22 dilemma, we present an innovative two-stage framework, GPTLENS, by designing the LLM to play two adversarial agent roles: auditor and critic. The auditor focuses on uncovering diverse vulnerabilities complemented by intermediate reasoning while the critic assesses the validity of these vulnerabilities and the associated reasoning. Empirical results demonstrate that GPTLENS delivers pronounced improvements over the conventional one-stage detection and is entirely LLM-driven, which negates the dependency for specialist expertise in smart contracts and exhibits generalization to a broad spectrum of vulnerabilities.

VIII. ACKNOWLEDGMENT

REFERENCES

[1] AlgoWriting. A simple guide to setting the gpt-3 temperature, 2020. https://algowriting.medium.com/gpt-3-temperature-setting-101-41200ff0d0be.

[2] Anthropic. Introducing claude. Anthropic Blog, 2022. https://www.anthropic.com/index/introducing-claude.

[3] L. Brent, N. Grech, S. Lagouvardos, B. Scholz, and Y. Smaragdakis. Ethainter: a smart contract security analyzer for composite vulnerabilities. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 454–469, 2020.

[4] L. Brent, A. Jurisevic, M. Kong, E. Liu, F. Gauthier, V. Gramoli, R. Holz, and B. Scholz. Vandal: A scalable security analysis framework for smart contracts. *arXiv preprint arXiv:1809.03981*, 2018.

[5] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

[6] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

[7] Y. Cao, S. Hu, Y. Gong, Z. Li, Y. Yang, Q. Liu, and S. Ji. Gift: Graph-guided feature transfer for cold-start video click-through rate prediction. In *Proceedings of the 31st ACM International Conference on Information & Knowledge Management*, pages 2964–2973, 2022.

[8] C. Chen, J. Su, J. Chen, Y. Wang, T. Bi, Y. Wang, X. Lin, T. Chen, and Z. Zheng. When chatgpt meets smart contract vulnerability detection: How far are we? *arXiv preprint arXiv:2309.05520*, 2023.

[9] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. d. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

[10] I. David, L. Zhou, K. Qin, D. Song, L. Cavallaro, and A. Gervais. Do you still need a manual smart contract audit? *arXiv preprint arXiv:2306.12338*, 2023.

[11] J. Feist, G. Grieco, and 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)*, pages 8–15. IEEE, 2019.

[12] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio. Generative adversarial nets. *Advances in neural information processing systems*, 27, 2014.

[13] G. Grieco, W. Song, A. Cygan, J. Feist, and A. Groce. Echidna: effective, usable, and fast fuzzing for smart contracts. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 557–560, 2020.

[14] J. Heller. *Catch-22: a novel*, volume 4. Simon and Schuster, 1999.

[15] X. Hou, Y. Zhao, Y. Liu, Z. Yang, K. Wang, L. Li, X. Luo, D. Lo, J. Grundy, and H. Wang. Large language models for software engineering: A systematic literature review. *arXiv preprint arXiv:2308.10620*, 2023.

[16] S. Hu, Z. Zhang, B. Luo, S. Lu, B. He, and L. Liu. Bert4eth: A pre-trained transformer for ethereum fraud detection. In *Proceedings of the ACM Web Conference 2023*, pages 2189–2197, 2023.

[17] S. Jeon, G. Lee, H. Kim, and S. S. Woo. Smartcondetect: Highly accurate smart contract code vulnerability detection mechanism using bert. In *KDD Workshop on Programming Language Processing*, 2021.

[18] B. Jiang, Y. Liu, and W. K. Chan. Contractfuzzer: Fuzzing smart contracts for vulnerability detection. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 259–269, 2018.

[19] S. Kalra, S. Goel, M. Dhawan, and S. Sharma. Zeus: analyzing safety of smart contracts. In *Ndss*, pages 1–12, 2018.

[20] Y. Li, D. Choi, J. Chung, N. Kushman, J. Schrittwieser, R. Leblond, T. Eccles, J. Keeling, F. Gimeno, A. Dal Lago, et al. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097, 2022.

[21] Z. Liao, Z. Zheng, X. Chen, and Y. Nan. Smartdagger: a bytecode-based static analysis approach for detecting cross-contract vulnerability. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 752–764, 2022.

[22] Z. Liu, P. Qian, X. Wang, L. Zhu, Q. He, and S. Ji. Smart contract vulnerability detection: from pure neural network to interpretable graph feature and expert pattern fusion. *arXiv preprint arXiv:2106.09282*, 2021.

[23] Z. Liu, P. Qian, X. Wang, Y. Zhuang, L. Qiu, and X. Wang. Combining graph neural networks with expert knowledge for smart contract vulnerability detection. *IEEE Transactions on Knowledge and Data Engineering*, 2021.

[24] M. Mossberg, F. Manzano, E. Hennenfent, A. Groce, G. Grieco, J. Feist, T. Brunson, and A. Dinaburg. Manticore: A user-friendly symbolic execution framework for binaries and smart contracts. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1186–1189. IEEE, 2019.

[25] Mythril. https://github.com/Consensys/mythril.

[26] OpenAI. Gpt-4 technical report, 2023. https://arxiv.org/abs/2303.08774.

[27] J. S. Park, J. C. O'Brien, C. J. Cai, M. R. Morris, P. Liang, and M. S. Bernstein. Generative agents: Interactive simulacra of human behavior. *arXiv preprint arXiv:2304.03442*, 2023.

[28] R. Paul, M. M. Hossain, M. Hasan, and A. Iqbal. Automated program repair based on code review: How do pre-trained transformer models perform? *arXiv preprint arXiv:2304.07840*, 2023.

[29] A. Permenev, D. Dimitrov, P. Tsankov, D. Drachsler-Cohen, and M. Vechev. Verx: Safety verification of smart contracts. In *2020 IEEE symposium on security and privacy (SP)*, pages 1661–1677. IEEE, 2020.

[30] C. Qian, X. Cong, C. Yang, W. Chen, Y. Su, J. Xu, Z. Liu, and M. Sun. Communicative agents for software development. *arXiv preprint arXiv:2307.07924*, 2023.

[31] P. Qian, Z. Liu, Q. He, R. Zimmermann, and X. Wang. Towards automated reentrancy detection for smart contracts based on sequential models. *IEEE Access*, 8:19685–19695, 2020.

[32] P. Qian, Z. Liu, Y. Yin, and Q. He. Cross-modality mutual learning for enhancing smart contract vulnerability detection on bytecode. In *Proceedings of the ACM Web Conference 2023*, pages 2220–2229, 2023.

[33] A. Radford, L. Metz, and S. Chintala. Unsupervised representation learning with deep convolutional generative adversarial networks. *arXiv preprint arXiv:1511.06434*, 2015.

[34] T. Schick, J. Dwivedi-Yu, R. Dessì, R. Raileanu, M. Lomeli, L. Zettlemoyer, N. Cancedda, and T. Scialom. Toolformer: Language models can teach themselves to use tools. *arXiv preprint arXiv:2302.04761*, 2023.

[35] C. Sendner, H. Chen, H. Fereidooni, L. Petzi, J. König, J. Stang, A. Dmitrienko, A.-R. Sadeghi, and F. Koushanfar. Smarter contracts: Detecting vulnerabilities in smart contracts with deep transfer learning. In *NDSS*, 2023.

[36] D. Shen, X. Chen, C. Wang, K. Sen, and D. Song. Benchmarking language models for code syntax understanding. *arXiv preprint arXiv:2210.14473*, 2022.

[37] S. So, M. Lee, J. Park, H. Lee, and H. Oh. Verismart: A highly precise safety verifier for ethereum smart contracts. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1678–1694. IEEE, 2020.

[38] X. Sun, L. Tu, J. Zhang, J. Cai, B. Li, and Y. Wang. Assbert: Active and semi-supervised bert for smart contract vulnerability detection. *Journal of Information Security and Applications*, 73:103423, 2023.

[39] Y. Sun and L. Gu. Attention-based machine learning model for smart contract vulnerability detection. In *Journal of physics: conference series*, volume 1820, page 012004. IOP Publishing, 2021.

[40] Y. Sun, D. Wu, Y. Xue, H. Liu, H. Wang, Z. Xu, X. Xie, and Y. Liu. When gpt meets program analysis: Towards intelligent detection of smart contract logic vulnerabilities in gptscan. *arXiv preprint arXiv:2308.03314*, 2023.

[41] W. J.-W. Tann, X. J. Han, S. S. Gupta, and Y.-S. Ong. Towards safer smart contracts: A sequence learning approach to detecting security threats. *arXiv preprint arXiv:1811.06632*, 2018.

[42] C. Thapa, S. I. Jang, M. E. Ahmed, S. Camtepe, J. Pieprzyk, and S. Nepal. Transformer-based language models for software vulnerability detection. In *Proceedings of the 38th Annual Computer Security Applications Conference*, pages 481–496, 2022.

[43] H. Touvron, T. Lavril, G. Izacard, X. Martinet, M.-A. Lachaux, T. Lacroix, B. Rozière, N. Goyal, E. Hambro, F. Azhar, et al. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*, 2023.

[44] P. Tsankov, A. Dan, D. Drachsler-Cohen, A. Gervais, F. Buenzli, and M. Vechev. Securify: Practical security analysis of smart contracts. In *Proceedings of the 2018 ACM SIGSAC conference on computer and communications security*, pages 67–82, 2018.

[45] N. vulnerability database. Common vulnerabilities and exposures (cves). https://cve.mitre.org/index.html.

[46] Y. Wang, H. Le, A. D. Gotmare, N. D. Bui, J. Li, and S. C. Hoi. Codet5+: Open code large language models for code understanding and generation. *arXiv preprint arXiv:2305.07922*, 2023.

[47] J. Wei, X. Wang, D. Schuurmans, M. Bosma, F. Xia, E. Chi, Q. V. Le, D. Zhou, et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in Neural Information Processing Systems*, 35:24824–24837, 2022.

[48] D. Wong and M. Hemmel. Decentralized application security project top 10 of 2018, 2018.

[49] Y. Wu, N. Jiang, H. V. Pham, T. Lutellier, J. Davis, L. Tan, P. Babkin, and S. Shah. How effective are neural networks for fixing security vulnerabilities. *arXiv preprint arXiv:2305.18607*, 2023.

[50] V. Wüstholz and M. Christakis. Harvey: A greybox fuzzer for smart contracts. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1398–1409, 2020.

[51] Z. Xi, W. Chen, X. Guo, W. He, Y. Ding, B. Hong, M. Zhang, J. Wang, S. Jin, E. Zhou, et al. The rise and potential of large language model based agents: A survey. *arXiv preprint arXiv:2309.07864*, 2023.

[52] C. S. Xia, Y. Wei, and L. Zhang. Automated program repair in the era of large pre-trained language models. In *Proceedings of the 45th International Conference on Software Engineering (ICSE 2023). Association for Computing Machinery*, 2023.

[53] Y. Xue, J. Ye, W. Zhang, J. Sun, L. Ma, H. Wang, and J. Zhao. xfuzz: Machine learning guided cross-contract fuzzing. *IEEE Transactions on Dependable and Secure Computing*, 2022.

[54] S. Yao, D. Yu, J. Zhao, I. Shafran, T. L. Griffiths, Y. Cao, and K. Narasimhan. Tree of thoughts: Deliberate problem solving with large language models. *arXiv preprint arXiv:2305.10601*, 2023.

[55] W. Zhang, S. Banescu, L. Pasos, S. Stewart, and V. Ganesh. Mpro: Combining static and symbolic analysis for scalable testing of smart contract. In *2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE)*, pages 456–462. IEEE, 2019.

[56] Y. Zhang, J. Yang, Y. Yuan, and A. C.-C. Yao. Cumulative reasoning with large language models. *arXiv preprint arXiv:2308.04371*, 2023.

[57] Z. Zhang, B. Zhang, W. Xu, and Z. Lin. Demystifying exploitable bugs in smart contracts. ICSE, 2023.

[58] W. X. Zhao, K. Zhou, J. Li, T. Tang, X. Wang, Y. Hou, Y. Min, B. Zhang, J. Zhang, Z. Dong, et al. A survey of large language models. *arXiv preprint arXiv:2303.18223*, 2023.

[59] L. Zhou, X. Xiong, J. Ernstberger, S. Chaliasos, Z. Wang, Y. Wang, K. Qin, R. Wattenhofer, D. Song, and A. Gervais. Sok: Decentralized finance (defi) attacks. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 2444–2461. IEEE, 2023.

[60] Y. Zhuang, Z. Liu, P. Qian, Q. Liu, X. Wang, and Q. He. Smart contract vulnerability detection using graph neural networks. In *Proceedings of the Twenty-Ninth International Conference on International Joint Conferences on Artificial Intelligence*, pages 3283–3290, 2021.

APPENDIX A
SOURCE CODE OF CASE STUDY

```solidity
1  pragma solidity ^0.4.24;
2
3  library SafeMath {
4      // ... (contents of the SafeMath library)
5  }
6
7  contract ERC20Basic {
8      uint public totalSupply;
9      function balanceOf(address who) constant returns (uint);
10     function transfer(address to, uint value);
11     event Transfer(address indexed from, address indexed to, uint value);
12     function allowance(address owner, address spender) constant returns (uint);
13     function transferFrom(address from, address to, uint value);
14     function approve(address spender, uint value);
15     event Approval(address indexed owner, address indexed spender, uint value);
16  }
17
18  contract BasicToken is ERC20Basic {
19      using SafeMath for uint;
20      address public owner;
21      bool public transferable = true;
22      mapping(address => uint) balances;
23      mapping(address => bool) public frozenAccount;
24      modifier onlyPayloadSize(uint size) {
25          if (msg.data.length < size + 4) {
26              throw;
27          }
28          _;
29      }
30      modifier unFrozenAccount {
31          require(!frozenAccount[msg.sender]);
32          _;
33      }
34      modifier onlyOwner {
35          if (owner == msg.sender) {
36              _;
37          } else {
38              InvalidCaller(msg.sender);
39              throw;
40          }
41      }
42      modifier onlyTransferable {
43          if (transferable) {
44              _;
45          } else {
46              LiquidityAlarm("The liquidity is switched off");
47              throw;
48          }
49      }
50      event FrozenFunds(address target, bool frozen);
51      event InvalidCaller(address caller);
52      event Burn(address caller, uint value);
53      event OwnershipTransferred(address indexed from, address indexed to);
54      event InvalidAccount(address indexed addr, bytes msg);
55      event LiquidityAlarm(bytes msg);
56      function transfer(address _to, uint _value) onlyPayloadSize(2 * 32)
              unFrozenAccount onlyTransferable {
57          if (frozenAccount[_to]) {
58              InvalidAccount(_to, "The receiver account is frozen");
59          } else {
60              balances[msg.sender] = balances[msg.sender].sub(_value);
61              balances[_to] = balances[_to].add(_value);
62              Transfer(msg.sender, _to, _value);
63          }
64      }
65      function balanceOf(address _owner) view returns (uint balance) {
66          return balances[_owner];
67      }
68      function freezeAccount(address target, bool freeze) onlyOwner public {
69          frozenAccount[target] = freeze;
70          FrozenFunds(target, freeze);
71      }
72      function accountFrozenStatus(address target) view returns (bool frozen) {
73          return frozenAccount[target];
74      }
75      function transferOwnership(address newOwner) onlyOwner public {
76          if (newOwner != address(0)) {
77              address oldOwner = owner;
78              owner = newOwner;
79              OwnershipTransferred(oldOwner, owner);
80          }
81      }
82      function switchLiquidity(bool _transferable) onlyOwner returns (bool success) {
83          transferable = _transferable;
84          return true;
85      }
86      function liquidityStatus() view returns (bool _transferable) {
87          return transferable;
88      }
89  }
90
91  contract StandardToken is BasicToken {
92      mapping(address => mapping(address => uint)) allowed;
93      function transferFrom(address _from, address _to, uint _value) onlyPayloadSize
              (3 * 32) unFrozenAccount onlyTransferable {
94          var _allowance = allowed[_from][msg.sender];
95          require(!frozenAccount[_from] && !frozenAccount[_to]);
96          balances[_to] = balances[_to].add(_value);
97          balances[_from] = balances[_from].sub(_value);
98          allowed[_from][msg.sender] = _allowance.sub(_value);
99          Transfer(_from, _to, _value);
100     }
101     function approve(address _spender, uint _value) unFrozenAccount {
102         if ((_value != 0) && (allowed[msg.sender][_spender] != 0)) throw;
103         allowed[msg.sender][_spender] = _value;
104         Approval(msg.sender, _spender, _value);
105     }
106     function allowance(address _owner, address _spender) view returns (uint
              remaining) {
107         return allowed[_owner][_spender];
108     }
109 }
110
111 contract BAFCToken is StandardToken {
112     string public name = "Business Alliance Financial Circle";
113     string public symbol = "BAFC";
114     uint public decimals = 18;
115     function UBSexToken() {
116         owner = msg.sender;
117         totalSupply = 1.9 * 10 ** 26;
118         balances[owner] = totalSupply;
119     }
120     function () public payable {
121         revert();
122     }
123 }
```

Listing 4. Smart contract code reported in CVE 2018-19830