Combining Fine-tuning and LLM-based Agents for Intuitive Smart Contract Auditing with Justifications

Wei Ma[1], Daoyuan Wu[2*], Yuqiang Sun[1], Tianwen Wang[3], Shangqing Liu[1], Jian Zhang[1], Yue Xue[4], Yang Liu[1]
[1] *Nanyang Technological University, Singapore, Singapore*
[2] *The Hong Kong University of Science and Technology, Hong Kong SAR, China*
[3] *National University of Singapore, Singapore, Singapore*
[4] *MetaTrust Labs, Singapore, Singapore*
ma_wei@ntu.edu.sg, daoyuan@cse.ust.hk, suny0056@e.ntu.edu.sg, tianwenw.vk@gmail.com,
liu.shangqing@ntu.edu.sg, jian_zhang@ntu.edu.sg, xueyue@metatrust.io, yangliu@ntu.edu.sg

***Abstract*—Smart contracts are decentralized applications built atop blockchains like Ethereum. Recent research has shown that large language models (LLMs) have potential in auditing smart contracts, but the state-of-the-art indicates that even GPT-4 can achieve only 30% precision (when both decision and justification are correct). This is likely because off-the-shelf LLMs were primarily pre-trained on a general text/code corpus and not fine-tuned on the specific domain of Solidity smart contract auditing.**

In this paper, we propose iAudit, a general framework that combines fine-tuning and LLM-based agents for intuitive smart contract auditing with justifications. Specifically, iAudit is inspired by the observation that expert human auditors first perceive what could be wrong and then perform a detailed analysis of the code to identify the cause. As such, iAudit employs a two-stage fine-tuning approach: it first tunes a Detector model to make decisions and then tunes a Reasoner model to generate causes of vulnerabilities. However, fine-tuning alone faces challenges in accurately identifying the optimal cause of a vulnerability. Therefore, we introduce two LLM-based agents, the Ranker and Critic, to iteratively select and debate the most suitable cause of vulnerability based on the output of the fine-tuned Reasoner model. To evaluate iAudit, we collected a balanced dataset with 1,734 positive and 1,810 negative samples to fine-tune iAudit. We then compared it with traditional fine-tuned models (CodeBERT, GraphCodeBERT, CodeT5, and UnixCoder) as well as prompt learning-based LLMs (GPT4, GPT-3.5, and CodeLlama-13b/34b). On a dataset of 263 real smart contract vulnerabilities, iAudit achieves an F1 score of 91.21% and an accuracy of 91.11%. The causes generated by iAudit achieved a consistency of about 38% compared to the ground truth causes.

> "One of the big skills in bug bounties that's really difficult to teach is intuition. Everything I do I am following my intuition. It's what looks interesting and what doesn't look right."
>
> — Katie Paxton-Fear
> One of the million-dollar-earning hackers [1].

I. Introduction

Smart contracts have emerged as a key application based on blockchain technology since the advent of Ethereum. Due to their openness, transparency, and irreversibility, smart contracts have become the foundation of decentralized financial applications (DeFi). However, since DeFi manages a significant amount of digital assets, identifying and fixing vulnerabilities in smart contracts is crucial. Currently, the real vulnerabilities exploited by hackers in smart contracts are mainly due to logical flaws [2], which render traditional pattern-based program analysis [3]–[10] less effective. According to Defillama Hacks [11], vulnerability attacks have caused losses of around $7.69 billion as of March 2024. Hence, there is an urgent need for innovative methods to combat these emerging threats.

Recent research [12]–[15] has shown that large language models (LLMs) have potential in auditing smart contracts, especially in demonstrating superior performance in detecting logic vulnerabilities [2], [13]. However, a recent systematic evaluation study [15] shows that even when equipping the LLM-based vulnerability detection paradigm with a state-of-the-art approach, namely enhancing GPT-4 with summarized vulnerability knowledge in a Retrieval Augmented Generation (RAG) [16] fashion, it still achieves only ∼30% precision when both the decision (i.e., whether the subject code is vulnerable) and justification (i.e., pinpointing the correct vulnerability type) are correct. This can be attributed to the fact that off-the-shelf LLMs (e.g., GPT-4), which were primarily pre-trained on a general text/code corpus, were not fine-tuned for the specific domain of Solidity[1] smart contract auditing.

Fine-tuning [17], [18] could be a promising approach to embed Solidity-specific vulnerability data into the model itself, compared to RAG [19], and thus address the problem mentioned above. In particular, by fine-tuning an LLM with vulnerable and non-vulnerable code, it could effectively *perceive* whether a new piece of code is vulnerable or not. According to insights from a million-dollar-earning hacker mentioned in the prologue, such intuition is quite important for vulnerability auditing. As such, instead of fine-tuning a single model to generate both vulnerability decisions (i.e., Yes or No) and the causes of vulnerabilities (i.e., the type or reason) simultaneously, we propose a novel two-stage fine-tuning approach. This approach first tunes a Detector model

* Corresponding author: Daoyuan Wu. Work conducted while at NTU.

[1] Solidity is a mainstream language for smart contract development.

to make decisions only, and then tunes a Reasoner model to generate the causes of vulnerabilities. In this way, the fine-tuned LLMs could mimic human hackers by first making intuitive judgments and then performing follow-up analysis of the code to identify the reasons for vulnerabilities.

We implement this "perception-then-analysis" fune-tuning into a general framework called iAudit[2] for intuitive smart contract auditing. In this implementation, iAudit allows Detector to make multiple intuitive judgments, each representing one perception. To achieve this, iAudit generates multiple variant prompts for the same vulnerability label to tune Detector and similarly employs multiple variant prompts for the same vulnerability reason to tune Reasoner. While it is possible to determine the optimal decision based on majority voting, fine-tuning alone cannot *identify the optimal cause for a vulnerability during the inference phase*. To address this new problem, we introduce the concept of LLM-based agents to the paradigm of fine-tuning in iAudit. Specifically, we introduce two dedicated LLM-based agents, the Ranker and Critic agents, to iteratively select and debate the most appropriate cause of vulnerability based on the output of the fine-tuned Reasoner model.

To obtain high-quality data for training and testing iAudit, we propose leveraging reputable auditing reports to collect positive samples and employing our own data enhancement method to derive negative samples. Eventually, we collected a balanced dataset consisting of 1,734 positive samples, i.e., vulnerable functions with reasons from 263 smart contract auditing reports, and 1,810 negative samples, i.e., non-vulnerable benign code. We then compared iAudit with traditional full-model fine-tuning methods, including CodeBERT, GraphCodeBERT, CodeT5, and UnixCoder, as well as with prompt learning-based LLMs, such as GPT-4/GPT-3.5 and CodeLlama-13b/34b. Our experimental results show that iAudit achieved an F1 score of 91.21%, significantly outperforming prompt learning-based LLMs (which are in the range of 60%+) and also notably beating other fine-tuned models (which are in the range of 80%+) that used the same training data as ours. Furthermore, in terms of alignment with ground-truth explanations, iAudit's output is clearly superior to that of other models, reaching a consistency rate of 37.99%. In contrast, the second-ranked GPT-4 achieves only 24%.

Besides the evaluation results, we also conducted three ablation studies to further justify iAudit's two-stage fine-tuning and majority voting strategies, as well as to measure the impact of additional call graph information on the model's performance. We summarize the key findings as follows:

- iAudit's two-stage approach achieved better detection performance than the integration model, which outputs labels and reasons simultaneously. We also experimentally confirmed that the model struggles to focus on the labels when required to output both types of information.
- Majority voting enhances the detection performance and stability. Using multiple prompts also allows the model to perform better than when using a single prompt.
- Call graph information may enable the model to make better judgments in some cases, but we also observed situations where this additional information could potentially confuse the model, thereby reducing its performance.

Roadmap. The rest of this paper is organized as follows. We first introduce the relevant background in Sec. II, followed by the design of iAudit in Sec. III. We then present our experimental setup and the results in Sec. IV. After that, we discuss related work and the limitations in Sec. V and Sec. VI, respectively. Finally, Sec. VIII concludes this paper.

Availability. To facilitate future research and comparison, we have made the inference code and dataset available at [20].

II. Background

A. Pre-trained Models and Large Language Models

Pre-trained models are models that have been initially trained on large datasets. These models can be quickly adapted to various specific tasks with minimal adjustments, avoiding the complex training process from scratch. Currently, most pre-trained models adopt an architecture based on transformers [21]. The innovation of this approach is that pre-trained models leverage large data and well-designed tasks for effective feature learning, which has been proven effective in multiple fields, such as text processing, image recognition, and software engineering. The standard transformer structure consists of one encoder and one decoder, which are structurally similar but function differently. Pre-trained models can be classified into encoder-based, decoder-based, or encoder-decoder combined types depending on the transformer structure used. For example, encoder-based models are represented by BERT [22] and CodeBERT [23], decoder models by the GPT series [24], [25], and encoder-decoder models by BART [26], T5 [27], and CodeT5 [28]. Compared with general pre-trained models, Large Language Models (LLMs) [29], [30] differ significantly in their used larger data and model scales. These models are trained by learning world-wide knowledge bases, typically reaching billions in scale. As the model size, data volume, and computational capacity increase, performance also improves, as revealed by the Scaling Laws [31]. Closed-source LLMs like GPT-3.5, GPT-4, and Gemini [32] offer their services externally through APIs, while open-source models like Llama2 [33] can achieve performance comparable or better to closed-source models after fine-tuning.

B. Parameter-Efficient Fine-Tuning

LLMs have extremely large parameters. Fully fine-tuning a large language model requires significant hardware resources and is very costly. Therefore, lightweight parameter fine-tuning [34], [35] is currently the main method of using LLMs compared to fully fine-tuning them. Although LLMs can be used without task-specific fine-tuning through in-context learning [36], this usually requires carefully prepared prompts. Furthermore, research has found that partial fine-tuning of

[2]iAudit is deployed as an auditing module of MetaTrust Labs' TrustLLM; see https://huggingface.co/MetaTrustSig.

LLMs with smaller parameters can achieve or even surpass the effects of huge models [37], [38]. These fine-tuning methods differ from full-model fine-tuning by focusing only on fine-tuning additional parameters while keeping the large model weights fixed, known collectively as parameter-efficient fine-tuning [34], [35]. They can be generally categorized into four types: Adapter [37], Low-Rank Adaptation (LoRA) [39], prefix tuning [40], and prompt tuning [41].

Adapter [37] adds a lightweight additional module to each layer of the model to capture information specific to downstream tasks. During optimization, only the parameters of the additional module are optimized. Since the number of parameters in the Adapter is much smaller than that of the model itself, it significantly reduces the overall parameter count and computational complexity of the model.

Low-Rank Adaptation (LoRA) [39] is a parameter-efficient adaptation method for LLMs, which adjusts LLMs for downstream tasks at a lower parameter cost. The core idea of LoRA is to introduce additional, low-rank adaptation parameters into the self-attention mechanism, effectively adjusting the model to suit new tasks with minimal addition of extra parameters.

Prefix tuning [40] adds a "prefix" sequence to each layer of the model, serving as additional context input. This method allows the model to adapt to specific tasks while retaining most of the knowledge acquired during pre-training. Unlike prefix tuning, prompt tuning [41] adds prompt tokens to the input, which can be placed at any position.

To sum up, using adapters can increase inference latency [39], [42]. Prefix or prompt tuning is subject to structural constraints that inhibit the learning of new attention patterns [43]. LoRA is an efficient method with low cost and can have a performance close to the full fine-tuning approach [39].

C. Smart Contracts and Their Vulnerabilities

Smart contracts are essential for realizing decentralized finance [44] as an application layer of blockchain technology. According to data from DeFiLlama [45], as of March 2024, the total value locked in the top three blockchain platforms (Ethereum, Tron, and BSC) has reached $73 billions. Given the close relationship between smart contracts and economic interests, their security has attracted widespread attention. Vulnerabilities in smart contracts can lead to significant losses, such reentrancy attacks and access-control attacks [46].

In the real world, hackers employ even more complex tactics. Currently, to address vulnerabilities in smart contracts, various static and dynamic detection tools [3]–[10] are used to test contract security. Unfortunately, some complex vulnerabilities are hard to be found by these detection tools. For example, in a sandwich attack [47], attackers monitor other pending transactions and execute their transactions first upon spotting a high-value yet uncompleted transaction. Due to this preemptive action, the attack transaction will be executed at a higher price, allowing the attacker to immediately sell the acquired excess profit for profit. Many well-funded project teams also invite third parties to audit their smart contracts before public release to ensure their safety.

III. Design of iAudit

As motivated in Sec. I, iAudit employs a novel two-stage fine-tuning approach and combines it with LLM-based agents for intuitive smart contract auditing with justifications. As shown in Fig. 1, iAudit has the following four roles:

- **Detector** is the key component for achieving intuitive smart contract auditing. By fine-tuning an LLM with vulnerable and non-vulnerable code, Detector can discern whether a piece of code is vulnerable, much like how a human hacker perceives a potential vulnerability.
- **Reasoner** takes the initial vulnerability perception from Detector to further analyze the potential causes of the vulnerability based on Detector's decision. By connecting Detector's output with Reasoner's reasoning during both training and inference, iAudit achieves two-stage fine-tuning.
- To identify the optimal cause of a vulnerability during the inference phase, we further introduce the concept of LLM-based agents into the fine-tuning paradigm in iAudit. Specifically, **Ranker** evaluates the reasons for each potential vulnerability, selecting a top explanation, while **Critic** further assesses Ranker's output to debate and determine the most appropriate cause of the vulnerability.

Challenges. While iAudit's four roles in Fig. 1 are intuitive, training and coordinating them well for effective smart contract auditing with reasonable justifications is difficult. More specifically, we encountered the following four challenges during the design and implementation of iAudit:

C1: *How to collect and derive high-quality training data?* For a fine-tuned model like iAudit, obtaining high-quality training data is always crucial. We propose leveraging reputable auditing reports to collect positive samples and employing our own data enhancement method to derive negative samples. Since this part is independent of iAudit's design, we defer its presentation to the end of this section in Sec. III-D.

C2: *How to make effective vulnerability judgements?* While fine-tuning a model with vulnerable and non-vulnerable code is straightforward, tuning it to be effective with limited data presents a challenge. We make an effort towards addressing this problem in Sec. III-A by opting to use multiple prompts for fine-tuning rather than a single prompt. The advantages of this approach are twofold: (i) it enriches the training dataset by increasing the volume of data, and (ii) it diminishes the bias associated with a single prompt, thereby enhancing the reliability of the results [48]. Optimal vulnerability perception could thus be achieved through majority voting.

C3: *How to effectively connect Detector's vulnerability sensing with Reasoner's vulnerability reasoning?* The fine-tuning of iAudit is unique because it employs a two-stage fine-tuning approach with the Detector and Reasoner models. Therefore, how to effectively connect these two models becomes a new

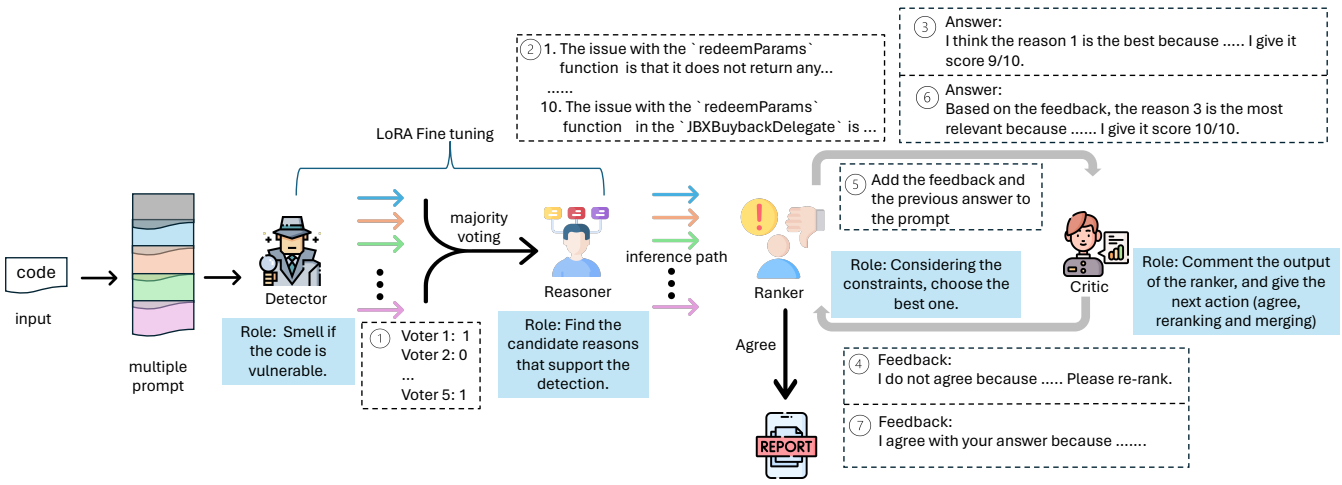

Fig. 1: An overview of iAudit, featuring its four roles: Detector, Reasoner, Ranker, and Critic.

issue not encountered in traditional fine-tuning. We present this aspect of iAudit's design in Sec. III-B.

C4: *How to obtain the optimal vulnerability cause from Reasoner's output?* Since Reasoner also employs multi-prompt fine-tuning, it is necessary to identify the optimal cause of vulnerability among the multiple causes output by Reasoner. We introduce two LLM-based agents, namely the Ranker and Critic components, in Sec. III-C, to iteratively select and debate the most appropriate cause of vulnerability.

An Example of Workflow. To wrap up, Fig. 1 also illustrates an example of iAudit's workflow. Initially, Detector perceives code vulnerabilities using five different inference paths (prompts). The perceived results are then subjected to majority voting to determine a consensus label. Based on the voting result, Reasoner interprets this outcome according to different inference paths, resulting in ten answers (each considering the context of the code location or not). Next, Ranker selects Reason 1 with a confidence score 9/10 and explains this choice. Critic challenges this choice and advises Ranker to re-evaluate. Taking Critic's feedback into account, Ranker re-ranks the ten reasons and selects Reason 3 with a confidence score of 10/10. Critic reviews Ranker's choice again and agrees with this decision. The loop is completed, and the final reason is returned to the user.

A. Using Multi-prompt Tuning and Majority Voting for Effective Vulnerability Judgements in the Detector

Detector is a fine-tuned expert model responsible for assessing whether code poses any risk. It mimics human intuitive judgment upon seeing a piece of code, assessing whether there are any issues. We employed LoRA [39] to fine-tune CodeLlama-13b [49] in the instruction manner [50] based on a high quality of dataset. During training, for the same input code, we wrap it with multiple prompts. These prompts, with different instruction formats, represent the different inference

Detector's Prompt Template

Below is an instruction that describes a classification task. [Task Description]
Instruction:
[Task Instruction]
Input:
[Input Description]:
"'Solidiy
{code}
"'
Response:

The label is {Label Name}.

Fig. 2: The Prompt Template Used by Detector.

paths, as illustrated in Fig. 1. In the inference phase of Detector, based on the output results of each prompt, we adopt a majority voting approach to determine the input label and use the voting ratio as the confidence score. Based on Detector's majority voting result, Reasoner in Sec. III-B then generates different reasons according to different inference paths.

It is worth noting that for the choice of the base model, we randomly selected 16 real logical vulnerabilities to evaluate three popular open-source models: StarCoder, Llama2, and CodeLlama. Upon manual review, we found that StarCoder sometimes refused to respond, and Llama2 provided one contradictory response with two labels. In contrast, CodeLlama offered a more stable response, which led us to choose CodeLlama as the foundational model for further fine-tuning.

The prompt template used by Detector is demonstrated in Fig. 2. Above the dashed line is the input x for our model. "{code}" is the placeholder for the input code. Below the dashed line, "The label is {Label Name}" is our target training output y, with "{Label Name}" being the label placeholder, which can be either "safe" and "vulnerable." The left ta-

ble (Detector's Multiple Prompts) in Fig. 4 details the [Task Description], [Task Instruction], and [Input Description], listed as notations *a*, *b*, and *c*, respectively. We fine-tune CodeLlama-13b using LoRA in a *generative* manner, as shown in Eq. 1,

$$L(\theta) = -\sum_{t=1}^{T} \log P_\theta(y_t|x, y_{<t}) \tag{1}$$

where θ represents the LoRA trainable parameters, T is the output sequence length, and $P_\theta(y_t|x, y_{<t})$ is the probability of the model with parameters θ generating a token y_t given the context x and all previous tokens $y_{<t}$. In this generative approach, the output at the time step t is conditioned only on the previous time steps ($< t$).

After fine-tuning, during inference, the proposed input follows the training format, and we need to extract labels from the output via keyword matching, using "safe" and "vulnerable." Since we employ multiple prompts, we obtain multiple label predictions and use majority voting to decide the final predicted label. In majority voting, each inference path casts a vote for one of the available labels, "safe" or "vulnerable," denoted as l_0 and l_1, respectively. The label that receives more than half of the total votes is declared the winner. Let $L = \{l_0, l_1\}$ represent the set of labels, and $V = \{v_1, v_2, \ldots, v_m\}$ represent the set of the prompt voter. Each prompt voter v_i casts a vote for one label. The winning label l_i is the one of l_0 and l_1 for which the following condition holds: $|\{v \in V : \text{vote}(v) = l_i\}| > \frac{m}{2}$. This condition asserts that the winning label o_w must receive more than half of the total votes m. We also use the voting ratio of the winning label as the confidence score for the final decision.

B. Connecting Detector for Reasoner's Tuning & Inference

Reasoner is an expert model responsible for reasoning about and explaining code vulnerabilities. It interprets the majority voting result of the Detector, generating multiple alternative explanations. During the Reasoner's LoRA fine-tuning process, our inputs include the code, its context, corresponding labels, and we construct zero-shot chain-of-thought (CoT) [51] prompts with different command formats for training. In the inference phase, Reasoner outputs multiple explanations based on the majority-voted label from Detector. We constructed two types of prompts: the first type includes the label, code information, and its function call relationships; the second type includes only the label and code information. In Sec. IV, we will investigate the impact of including function call relationships or not. For each type, we designed five different instruction formats for the prompts, totaling ten inference paths, as illustrated in Fig. 1.

The prompt template used by the Reasoner is shown in Fig. 3. "code", "caller info", "callee info", and "Target Reason" are placeholders for the input code, caller context, callee context, and the target output, respectively. The right table (Reasoner's Multiple Prompts) in Fig. 4 details the first prompt type with calling context, including [Task Description] denoted as *a*, [Task Instruction] denoted as *b*, [Input Description] denoted as *c*, [Caller Description] and [Callee Description] denoted as *d*, and [Label Information + Zero-shot-CoT Tip] denoted as *e*. For the second prompt type, the calling context is omitted. Reasoner employed the same fine-tuning method as Detector, as shown by Eq. 1. During inference, the proposed input prompt follows the training format, and Reasoner generates ten answers to interpret Detector's assessment.

Reasoner's Prompt Template

Below is an instruction that describes a reasoning task.
[Task Description]
Instruction: [Task Instruction]
Input:
[Input Description]:
"'Solidiy
{code}
"'
As a Caller: (Optional)
[Caller Description] "'
{caller info}
"'
As a Callee: (Optional)
[Callee Description]
"'
{callee info}
"'
Response:
[Label Information + Zero-shot-CoT Tip]

{Target Reason}

Fig. 3: The Prompt Template Used by Reasoner.

C. Ranking and Debating the Optimal Vulnerability Cause

Ranker and Critic are two LLM-based agents collaborating to select the most appropriate cause of vulnerability from multiple explanations returned by Reasoner for a given code function. Ranker performs two actions: "*rank*" and "*merge*". "*Rank*" involves selecting the best explanation from the ones provided, while "*merge*" involves integrating multiple selected explanations. We define 10 constraints for Ranker to select the top explanation. Critic evaluates Ranker's answer in conjunction with the code function, providing three next-step action instructions: "*agree*", "*rerank*", and "*merge*". "*Agree*" means the current answer is reasonable and can be returned to the user. "*Rerank*" indicates that Ranker needs to re-select, considering Critic's feedback and previous answers. "*Merge*" suggests that the top reasons provided must be integrated.

More specifically, Ranker employs the following 10 constraints in the prompt as its selection criteria:

1) If one reason describes code that does not exist in the provided input, it is not valid.
2) If one reason is not related to the code, the reason is not valid.
3) If this reason violates the facts, the reason is unreasonable.

	Detector's Multiple Prompts
1	a. Devise a label name suitable for categorizing items as either vulnerable or safe. b. Please review the code. Please find out if it is vulnerable. c. The function {fn_name} from the contract {contract_name}.
2	a. Suggest a label designation that clearly identifies an item's status as either vulnerable or safe. b. Inspect the following Solidity code. Determine if there are any vulnerabilities present. c. Observe the method {fn_name} within the smart contract {contract_name}.
3	a. Invent a naming label that aptly segregates items into vulnerable or safe classifications. b. Examine this Solidity script. Identify any potential security risks. c. Review the function {fn_name} in the blockchain contract {contract_name}.
4	a. Formulate a label descriptor that bifurcates objects into categories of vulnerable and safe. b. Please assess the provided Solidity code for any security vulnerabilities. c. Check the procedure {fn_name} in the digital contract {contract_name}.
5	a. Propose a label nomenclature that aptly differentiates between vulnerable and safe states. b. Evaluate the given Solidity function. Are there any security flaws? c. Inspect the subroutine {fn_name} from the decentralized contract {contract_name}.

	Reasoner's Multiple Prompts (for the type including function call relationships)
1	a. Examine the underlying factors and suggest a reason given the label name. b. Please review the code and its calling relationships as the caller and the callee. Given the label name, please find out the reason. c. The function {fn_name} from the contract {contract name}. d. {fn_name} calls these functions. {fn_name} is called by these functions. e. The input code is {label_name}. Please state the reason. Let's think step by step
2	a. Carefully assess the contributing factors and their interplay. Utilize the label name to form a coherent reasoning. b. Please analyze the code function and its dependencies, including both incoming and outgoing calls. Considering the label name, identify the underlying cause. c. The method {fn_name} in the smart contract {contract name}. d. Functions called by {fn_name}. Functions calling {fn_name}. e. Given that the code is labeled {label_name}, let's determine the reason by breaking down the process.
3	a. Delve into the core aspects and their significance. Use the label name to draw an informed inference. b. Examine the code's logic flow. Based on the label name, deduce the primary reason. c. The method {fn_name} in the smart contract {contract name}. d. d. External routines invoked by {fn_name}. Routines that invoke {fn_name}. e. With {label_name} as the code's label, let's systematically uncover the rationale.
4	a. Scrutinize the main factors and deduce a reason in light of the label name. b. Evaluate the code's connections and its purpose within the system. Using the label name, infer the main reason. a. Code segment {fn_name} from the blockchain contract {contract name}. b. Functions triggered by {fn_name}. Functions that trigger {fn_name}. c. Considering {label_name} as the designated label, let's sequentially analyze the reason.
5	a. Explore the root causes and provide a justification considering the assigned label name. b. Investigate the role and relationships of the code. Utilizing the label name, propose a probable reason. c. Procedure {fn_name} in the decentralized application {contract name}. Operations executed by {fn_name}. d. Operations that execute {fn_name}. e. Recognizing {label_name} as the code's label, let's logically deduce the reason.

Fig. 4: Detailed Multiple Prompts for Detector and Reasoner.

4) If one reason is not related to the decision, the reason is not valid.
5) If one reason assume any information that is not provided, the reason is not valid.
6) If the code is safe and one reason supports the decision, please check if the code has other potential vulnerabilities. If the code has other potential vulnerabilities, the reason is not valid.
7) The selected reason should be the most relevant to the decision.
8) The selected reason must be the most reasonable and accurate one.
9) The selected reason must be factual, logical and convincing.
10) Do not make any assumption out of the given code.

Both Ranker and Critic are LLMs agents implemented based on the Mixtral 8x7B-Instruct [52] model, the capability of which is close to that of larger LLMs [52]–[54]. Moreover, we have observed that the Mixture of Experts (MoE) [52] model can more effectively output data in the predetermined format than other models, making it easier for us to handle the output.

D. High-quality Training Data Collection and Enhancement

The quality of training data is crucial for fine-tuning LLMs. To collect positive samples, namely risky vulnerability code, we can employ auditing reports from reputable industry companies, such as Trail of Bits, Code4rena, and Immunefi. Specifically, we crawled and parsed 1,734 vulnerable functions with reasons from 263 smart contract auditing reports, which were assembled by a popular auditing website called Solodit [55].

However, to train our model, we also need non-vulnerable benign code (i.e., negative samples), but this type of data is missing in the audit reports. Therefore, we propose our own data enhancement method to derive high-quality negative samples. Specifically, we adopt the GPT-4-based approach described in LLM4Vuln [15] to extract vulnerability knowledge from vulnerability reports on Code4rena. This includes the functionality descriptions of vulnerable functions and the code-level reasons why the vulnerabilities occur. We then cluster this raw vulnerability knowledge based on the functionality descriptions into groups using Affinity Propagation [56] as described in [57] and use GPT-4 to summarize a functionality description for each group. With the hierarchical information of group functionality, individual functionality, and vulnerability negligence, we employ the hierarchical GPT-based matching (i.e., matching the group first, then matching functionality and negligence) in GPTScan [13] to obtain the label information for tested code. A function is labeled as a negative sample if no vulnerability information matches. All prompts used are from LLM4Vuln and GPTScan.

Eventually, we collected a balanced dataset with 1,734 positive samples and 1,810 negative samples. In this dataset, vulnerable functions have a median of 49.5 lines of code and

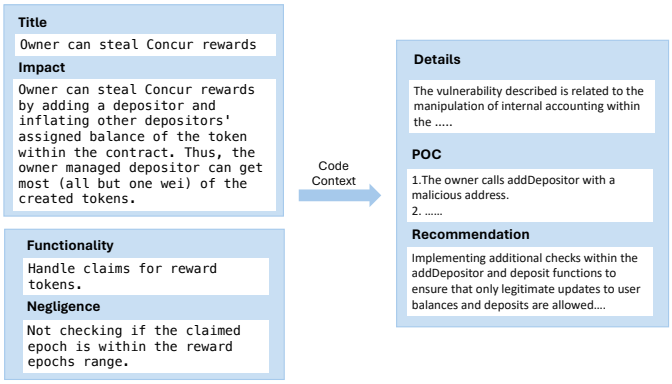

Fig. 5: Data Enhancement for Expanding Vulnerability Explanations based on GPT-3.5.

a complexity of 18.5, while safe functions have a median of 35.5 lines of code and a complexity of 13.5. The complexity distributions between the two types are somewhat similar but with a slight difference, indicated by a Kullback-Leibler (KL) divergence value of 0.1. This dataset was divided into **training**, **validation**, and **test** subsets, containing **2,268**, **567**, and **709** entries, respectively. During training, we use the training and validation sets. During testing, we use the test set.

After collecting the labeled and unlabeled data, we also obtained corresponding explanations for the vulnerabilities. However, the quality of these vulnerability justifications varies considerably. Furthermore, some data contain external links, which may cause the model to hallucinate and output non-existent links. To improve the interpretability of the reasons behind the vulnerabilities in the dataset, we used GPT-3.5 to enhance the existing explanations, expanding on the explanations of the vulnerabilities, proofs of concept (PoC), and recommended fixes. Fig. 5 shows an example, where the left part presents the original reasons for the vulnerability, which are short and lack detail. We thus use them as prompts, along with the code context, to instruct GPT-3.5 to generate more detailed descriptions, including the PoC and the mitigation recommendation.

Note that we chose GPT-3.5 mainly because it achieves a good balance between cost and effectiveness—much cheaper than GPT-4, yet comparable in quality for this specific task during our manual comparison. We also added constraints to our prompts during this process to ensure that the enhanced explanations aligned with the actual vulnerabilities.

IV. Evaluation

A. Experimental Setup

In our study, we carefully selected a series of benchmark models, categorized into two groups: LLMs for zero-shot learning and pre-trained code models based on fine-tuning, to ensure a comprehensive and sound comparative analysis. For zero-shot learning LLMs, we chose CodeLlama-13b-Instruct, CodeLlama-34b-Instruct [49], GPT-3.5, and GPT-4 as benchmarks, representing the current state-of-the-art. Additionally, we selected CodeBERT [23], GraphCodeBERT [58], CodeT5 [28], UnixCoder [59], and CodeLlama-13b [49] to train classifiers. Among these, CodeBERT, GraphCodeBERT, CodeT5, and UnixCoder underwent a complete model fine-tuning process to adapt to the specific code classification task. In particular, CodeLlama-13b employs LoRA for lightweight tuning and uses the last token representation for classification. Note that that our method is different; iAudit's Detector achieves classification by generating label names as task outputs.

B. Research Questions (RQs)

Since our proposed method comprises two core functions: vulnerability detection and reason explanation, we designed a series of experiments to evaluate and demonstrate the performance and effectiveness of both tasks. These experiments aim to answer the following research questions (RQs):

a) RQ1 - Performance Comparison: **How does the performance of iAudit in detecting vulnerabilities compare to other models?** This question aims to understand how the effectiveness of Detector in detecting vulnerabilities compares to that of other existing models. The focus is on comparative analysis, involving metrics accuracy, precision, recall, and F1 score, to evaluate and contrast the performance.

b) RQ2 - Explanation Alignment: **To what extent do the explanations generated by iAudit's Reasoner align with the real reasons?** RQ2 concerns the quality of the explanations the Reasoner provided for the decision of the Detector. It questions whether the reasons given by iAudit correspond to the actual reasons behind the vulnerabilities, emphasizing the interpretability and trustworthiness of the model.

c) RQ3 - Two-stage Approach vs. An Integration Model: **How does iAudit compare with an integration model that performs detection and reasoning simultaneously?** Our method is based on a generative model, with two models trained on the generated labels and reasons, respectively. Another approach uses a single model to generate both reasons and labels. This question explores the effectiveness and impact of integrating the Detector and Reasoner components into one.

d) RQ4 - Effectiveness of Majority Voting: **Can majority voting improve the effectiveness of the Detector?** RQ4 investigates whether the effectiveness of the Detector can be improved by adopting a majority voting mechanism. Majority voting, a technique that makes the final decision based on the majority output of multiple models, may improve the robustness and accuracy of the method.

e) RQ5 - Impact of Additional Information: The **call graph** illustrates the interaction of code with other components within the project, which is expected to be advantageous for our task. We address two specific research sub-questions:

- RQ5.1. Can the call graph enhance the Detector performance?
- RQ5.2. In what way does the call graph influence our explanation generation process, specifically within the Reasoner-Ranker-Critic pipeline?

TABLE I: Performance comparison between iAudit's Detector and zero-shot LLMs.

	F1	Recall	Precision	Accuracy
GPT-4	0.6809	**1**	0.5162	0.5162
GPT-3.5	0.6809	**1**	0.5162	0.5162
CodeLlama-13b	0.6767	0.9781	0.5173	0.5176
CodeLlama-34b	0.6725	0.9454	0.5219	0.5247
iAudit	**0.9121**	0.8934	**0.9316**	**0.9111**

TABLE II: Performance comparison between iAudit's Detector and other fine-tuned models.

	F1	Recall	Precision	Accuracy
CodeBERT	0.8221	0.7322	0.9371	0.8364
GraphCodeBERT	0.8841	0.8333	0.9414	0.8872
CodeT5	0.8481	0.7705	**0.9431**	0.8575
UnixCoder	0.8791	0.8443	0.9169	0.8801
CodeLlama-13b-class	0.8936	0.8716	0.9167	0.8928
iAudit	**0.9121**	**0.8934**	0.9316	**0.9111**

Besides the RQs above, we also used our model to audit two bounty projects (currently anonymous) on Code4rena. We invited audit experts to verify our findings. In the end, we found 6 critical vulnerabilities, which were recognized by the project team or audit experts. In particular, one vulnerability was not discovered by any tools, marked as a great finding. This demonstrates the real-world value of iAudit. Due to page limitations, we have included these case studies in the supplementary materials for interested readers.

C. RQ1 - Performance Comparison

Firstly, we compared iAudit with LLMs based on zero-shot learning, as shown in Table I. Our method also uses a zero-shot approach during the inference phase. We considered two proprietary models (GPT-4 and GPT-3.5) and three open-source models (CodeLlama-13b and CodeLlama-34b). For the open-source models, we strictly adhered to their prompt formats. Huggingface Transformer [60] has integrated these prompt formats into its framework. The format conversion is completed by calling `apply_chat_template`. CodeLlama requires adding [INST] and [/INST] as well as special tags «SYS» and «/SYS». As shown in Table I, after fine-tuning, our proposed strategy significantly outperforms the baseline models in the zero-shot scenario in terms of F1, accuracy, and precision, achieving high scores of 0.9121, 0.8934, and 0.9111, respectively. However, when examining the recall, we notice that, the baseline models all performed excellently. Notably, GPT-4 and GPT-3.5 achieved a recall score of 1. We checked the confusing matrix and found that all test data are labelled by the vulnerability. For GPT-4 and GPT-3.5, we adopted the prompts which are provided by our industrial partner, MetaTrust Labs, a Web3 security company.

Secondly, we compared Detector with fine-tuned models in detecting vulnerabilities, using F1, recall, precision, and accuracy as evaluation metrics, as shown in Table II. We compared our method with CodeBERT, GraphCodeBERT, CodeT5, UnixCoder, and CodeLlama-13b-class. CodeBERT, GraphCodeBERT, CodeT5, and UnixCoder underwent full model fine-tuning. These traditional pre-trained models use the first token of the input sequence as the feature input for the classifier. CodeBERT is based on the transformer encoder. GraphCodeBERT has the same architecture as CodeBERT but includes additional pre-training on data dependency relations. CodeT5 utilizes the transformer encoder and decoder, adopting an architecture similar to T5 [27]. UnixCoder unifies the encoder and decoder architecture, controlling the model behaviour through a masked attention matrix with prefix adapters. CodeLlama-13b-class performs classification based on LoRA. We fine-tuned CodeLlama-13b-class for LoRA classification using the PEFT framework. CodeLlama-13b-class uses the representation of the last token of the input sequence as the feature input for the classifier.

As shown in Table II, iAudit achieves the highest scores of F1, Recall, and Accuracy among all methods, 0.9121, 0.8934 and 0.9111. CodeLlama-13b-class is second only to our method regarding vulnerability detection rate, and the performance is relatively close. GraphCodeBERT and UnixCoder perform worse than CodeLlama-13b-class. Although CodeT5 achieves the highest precision at 0.9431, its other metrics are lower than GraphCodeBERT and UnixCoder. CodeBERT has the worst performance. Additionally, the accuracy scores of these models are relatively high (all are more than 0.91), indicating that many of the predicted risky vulnerabilities are indeed risky.

Regarding precision, iAudit produce more false positives than CodeT5, GraphCodeBERT, and CodeBERT. This is primarily because iAudit uses a longer context length (16k) compared to these models, whose 512-length context truncation causes the vulnerable and safe samples to align more closely in length, thus affecting the apparent code complexity. However, our dataset shows that vulnerable code is relatively more complex than safe code. This leads iAudit to output 10 unique false positives with more complex code.

> **Answer for RQ1:** The performance of iAudit's Detector exceeds that of traditional full-model fine-tuning, LoRA fine-tuning in a classification manner, and LLMs based on in-context learning. The performance of fine-tuned models is also better than that of zero-shot learning.

D. RQ2 - Explanation Alignment

To measure the effectiveness of Reasoner in explaining vulnerabilities, we compared the consistency between the explanations we generated and the root causes. Given the LLM's outstanding performance in interpreting textual meaning, we used GPT-4 to verify whether our generated explanations align with the root causes. For this consistency assessment, we employed automated annotation prompts from Y. Sun et al. [15]. The results of our consistency test are depicted in Fig. 6, where the y-axis represents the percentage of our explanations in the test set that match the root causes. Our method significantly outperformed the baseline methods, achieving a consistency

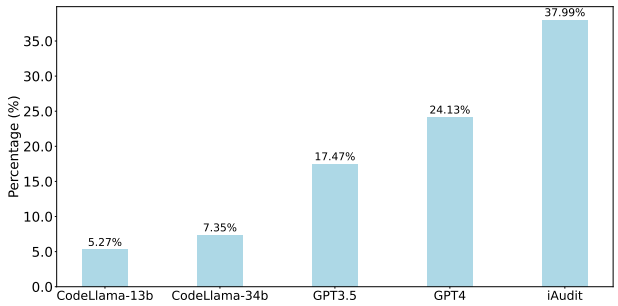

Fig. 6: Comparing the alignment with ground-truth reasons.

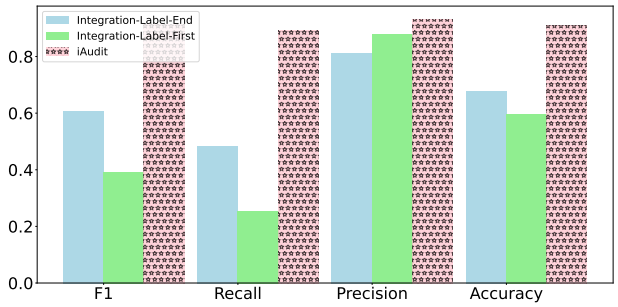

Fig. 7: Comparing iAudit with the integration models that make decisions and explain the vulnerabilities simultaneously.

rate of 37.99%, while no baseline method exceeded 25%. Among these baselines, GPT-4 performed the second best with 24.13% consistency. Additionally, the results also indicated that CodeLlama-13b had the weakest performance.

> **Answer for RQ2:** The rationality of Reasoner's output is clearly superior to that of other models. On the test set, its consistency with real reasons reaches 37.99%, which is over 10% higher than the second-ranked GPT-4.

E. RQ3 - Two-stage Approach vs. An Integration Model

Our research methodology involves vulnerability detection and explanation, executed in two stages. We trained two models, i.e., Detector and Reasoner, based on a generative approach to perform these tasks on their respective high-quality datasets. A question arises whether these two tasks can be merged and trained simultaneously in a single model. In response, we developed an integration model that generates labels and explanations for the vulnerabilities, comparing it to our two-stage approach. The integration model uses prompts similar to those of Reasoner, with additional requirements to output the label. We explored two integration training approaches: 1) generating labels first, then explaining the reasons; 2) explaining the reasons first, then generating labels.

The results, as shown in Fig. 7, indicate that our step-wise approach outperforms the integration models across four key performance metrics. While the three methods are similar in precision, the differences in other metrics are notable. Analyzing these results, we found that the integration methods have higher accuracy for negative samples but a lower detection rate for positive samples (i.e., lower recall). This may be attributed to the generative loss optimization, where the output sequence is longer, making the label-related loss occupy a smaller proportion of the total loss, thus preventing the model from adequately focusing on the label. To test this hypothesis, we added data that includes only label generation to the dataset during the integration training process, guiding the model to focus more on the label. In the evaluation phase, we still required the model to output both labels and explanations simultaneously. Through this mixed training approach, we observed a significant improvement in the model's vulnerability detection performance, with an F1 score of 0.8433, a recall rate of 0.8164, a precision of 0.8723, and an accuracy of 0.8434.

> **Answer for RQ3:** iAudit achieved better detection performance than the integration model that outputs labels and reasons simultaneously. We confirmed that the model struggles to focus on the labels when required to output both types of information, as evidenced by our inclusion of label-only data in the verification process.

TABLE III: Majority Voting vs. Single Prompt.

	F1	Recall	Precision	Accuracy
Single-prompt	0.8278	0.8005	0.8567	0.8279
Prompt-1	0.8988	0.8852	0.9127	0.8970
Prompt-2	0.9027	0.8743	**0.9329**	0.9027
Prompt-3	0.9063	0.8852	0.9284	0.9055
Prompt-4	0.9098	**0.8962**	0.9239	0.9083
Prompt-5	0.9096	0.8934	0.9263	0.9083
iAudit	**0.9121**	0.8934	0.9316	**0.9111**

F. RQ4 - Effectiveness of Majority Voting

Our research explored a method using multiple prompts and a voting mechanism for Detector to determine the final label. This method aims to enhance the model's precision and credibility. During the evaluation process, we continued to use metrics such as the F1 score, recall, precision, and accuracy. We calculated these metrics for each prompt individually for comparative analysis, as shown in Table III. It should be noted that the first row *Single-prompt* indicates that we used only one prompt format to train Detector. Prompt-1, Prompt-2, Prompt-3, Prompt-4, and Prompt-5 represent the results for each prompt after multiple-prompt training. The last row shows the results after majority voting, indicating that majority voting can improve the overall performance of iAudit, with both the F1 score and accuracy being the highest. At the same time, except for Single-prompt, we noticed minimal performance differences among multiple prompts. Single-prompt performed much worse than the others. Training with multiple prompts can improve model performance compared to using only one prompt during training.

Additionally, we divided the test set into two groups based on whether the predictions were correct or incorrect, named

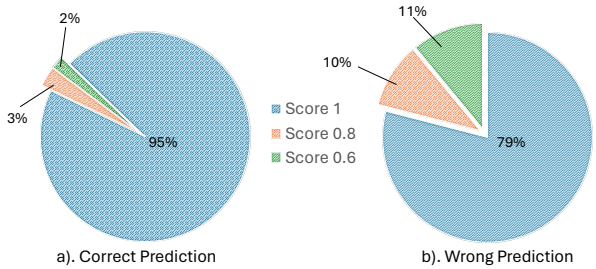

Fig. 8: The Distribution of Voting Scores for Correct Predictions and Wrong Predictions.

"correct prediction" and "incorrect prediction" groups, respectively, and analyzed the distribution of confidence scores within these two groups. We found that in the incorrect prediction group, the proportion of confidence scores within the range of 0.6 to 0.8 is significantly higher than in the correct prediction group (11% vs 2%, 10% vs 3%, respectively), as shown in Fig. 8. The confidence score can reflect the reliability of the prediction results to a certain extent. When the confidence score is low, the prediction results are less credible.

> **Answer for RQ4:** Majority voting enhances the detection performance and stability. Additionally, using multiple prompts allows the model to perform better and be more reliable than when using a single prompt.

G. RQ5 - Impact of Additional Information

We explored whether introducing additional call graph information into the model could enhance its performance. We added function call relationships to the prompts as contextual information.

a) RQ5.1: Through comparative experiments as shown in Table IV, we found that this calling contextual information did not improve the model's overall performance. In the second row, *Call*, we used the prompts with the calling information and then employed majority voting to decide the prediction result. For the third row, *Call-OutCall*, we used prompts both with and without calling information and also used majority voting. Compared with iAudit's Detector, they exhibited lower precision, accuracy, and f1, with almost the same recall.

b) RQ5.2: Fig. 9 demonstrates the selected reason distribution from Ranker-Critic. We can see that the majority (65%) of the selected reasons are from the prompts with calling information, while there is still a high ratio (35%) of selected reasons from the prompts without calling information.

Although function call relationships provide more information, this information does not always help the model better complete the current task. In some cases, this information may cause interference, making it difficult for the model to identify critical information, thereby resulting in more false positives and affecting performance. Furthermore, not all function call relationships are practically valuable. If these additional pieces of information are not closely related to the problem the model is trying to solve, they may not help enhance the model's performance. Our research indicates that merely adding function call information does not directly facilitate the model's effectiveness in detecting vulnerabilities. In the field of vulnerability detection, exploring how to construct effective contextual information remains a challenging and worthy research question.

TABLE IV: Impact with or without Additional Information.

	F1	Recall	Precision	Accuracy
Call	0.9011	**0.8962**	0.9061	0.8984
Call-OutCall	0.9083	0.8934	0.9237	0.9069
iAudit	**0.9121**	0.8934	**0.9316**	**0.9111**

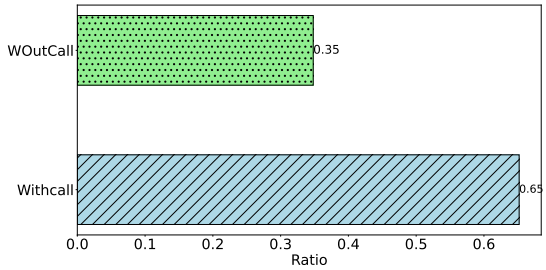

Fig. 9: Final Reason Distribution of Ranker-Critic.

> **Answer for RQ5:** Additional call graph information may enable the model to make better judgments in some cases. However, we also observed situations where this additional information could potentially confuse the model, thereby reducing its performance.

V. Related Work

Vulnerability detection has been a critical issue for the healthy and sustainable development of the software ecosystem, especially in blockchain and smart contracts. Traditional vulnerability detection methods, such as those based on predefined static analysis rules [3], often lack robust generalization capabilities and are difficult to adapt to new types of vulnerabilities. Moreover, some logic-related vulnerabilities [2] are also challenging to encapsulate into static analysis rules. To address this issue, researchers have employed deep learning-based approaches. For example, Zhuang et al. [61] used graph neural networks to detect vulnerabilities in smart contracts. Liu et al. [62] combined interpretable graph features with expert patterns to achieve better results and interpretable weights. Wu et al. [63] utilized a pre-training technique and critical data flow graphs for the detection of smart contract vulnerabilities.

With the advent of large language models, researchers are not only utilizing traditional deep learning models but also LLMs for vulnerability detection. For example, Ullah et al. [64] evaluated LLMs on vulnerability detection tasks and found that they may not perform well. Fu et al. [65]

further analyzed the gap for LLMs in detecting vulnerabilities. Thapa et al. [66] leveraged LLMs for software vulnerability detection, and David et al. [12] used LLMs for smart contract vulnerability tasks. Alqarni et al. [67] fine-tuned the BERT model [68] for source code vulnerability detection. Sun et al. [15] proposed an unified evaluation framework, LLM4Vuln, to enhance the ability of LLMs to detect vulnerabilities. Some research also fused large language models with traditional program analysis methods. Sun et al. [13] proposed GPTScan for smart contracts, leveraging static program analysis to reduce the false positives of LLMs. Li et al. [69] proposed LLift for integrating LLMs with static analysis tools. SmartInv [70] and PropertyGPT [71] further integrated large language models with formal verification methods, aiming not only to detect vulnerabilities but also to prove that a piece of code is secure.

However, all these studies have not tuned domain-specific knowledge into the models themselves, focusing only on the knowledge from the pre-training dataset or the vulnerable code segment itself, which could not effectively detect logic bugs.

VI. Threats to Validity

In the data collection process, there is a risk of data bias, which might prevent models trained and tested on these data from generalizing accurately. Moreover, the precision of data labelling significantly impacts model performance. To mitigate these issues, we collected verified data from real and public audit reports and utilized the latest tools, such as GPTScan [13] and LLM4Vuln [15], to assist in cleaning and annotating the data. It is important to note that external links in the data could induce LLM to produce incorrect information; therefore, we performed data cleaning to remove these links. Considering LLMs' sensitivity to input data, we standardized the code data, including removing unnecessary spaces without changing code semantics, to enhance the model robustness and reliability. To maximize the performance of the zero-shot learning of GPT-3.5 and GPT-4, we adopted and optimized the prompts from our partner, MetaTrust Labs. These prompts have been integrated into their working pipeline. For open-source models, we collaborated with an auditing expert to adapt their prompts for these models.

Overfitting is a common issue during model training, which we addressed by implementing an early stopping strategy. The choice of different models might affect the ranker-critic architecture's effectiveness. We tested multiple cutting-edge open-source models, including MoE [52], CodeLlama-70b [49], Llama2-70b [33], and the recently introduced Gemma [72], and compared their performance on inference benchmark tests. Based on factors like the strictness of the model output format and operational speed, we chose MoE [52]. Our research also showed that the consistency between the selected reasons from the MoE and the real reasons reached about 38%. To control costs, we limited the maximum iterations in the ranker-critic loop to five and adopted four-decimal precision handling.

VII. Limitation

Although our method performed excellently in trials, its primary advantage lies in detecting logic vulnerabilities in smart contracts, which account for more than 80% of exploitable vulnerabilities according to a recent study [2]. As such, while iAudit's training data does include instances of Reentrancy, Overflow, and Underflow vulnerabilities, handling them is not the major usage scenario of iAudit. Indeed, these traditional contract vulnerabilities are theoretically more suitable for detection by program analysis methods. That said, both AI-based and PL-based methods have their unique comfort zones, and since iAudit is fully AI-based, this paper compared it with other AI-based methods only. Additionally, despite our efforts to mitigate the phenomenon of hallucinations in large language models (LLMs) through voting and agents, hallucinations may still occur. Finally, since our tools rely on LLMs, there are certain hardware requirements. While the models can be compressed to run on less powerful GPUs, this compression may result in reduced model performance.

VIII. Conclusion

In this paper, we proposed iAudit, the first smart contract auditing framework that combines fine-tuning and LLM-based agents to detect vulnerabilities and explain the results. We adopted a multiple-prompt-based strategy and applied LoRA-based fine-tuning to train the Detector and Reasoner. The former generates results based on a majority voting mechanism, while the latter provides multiple alternative explanations based on different inference paths. Furthermore, we introduced two LLM agents, Ranker and Critic, to collaborate in selecting the most appropriate explanation. Our approach demonstrated superior performance in zero-shot scenarios compared to zero-shot LLM learning and traditional full-model fine-tuning methods. We studied the performance improvement brought by the majority voting strategy and compared different LoRA training methods, providing the rationality of our choice. We also explored how additional calling context affects our model's performance. For future work, we will focus on enhancing the model's stability and its alignment with human preferences.

Acknowledgment

We thank all the reviewers for their detailed and constructive comments. We also express our gratitude to all colleagues at MetaTrust Labs for their assistance in deploying the TrustLLM's iAudit model. This research/project is supported by the National Research Foundation, Singapore, and the Cyber Security Agency under its National Cybersecurity R&D Programme (NCRP25-P04-TAICeN), the National Research Foundation, Singapore, and DSO National Laboratories under the AI Singapore Programme (AISG Award No: AISG2-GC-2023-008), and NRF Investigatorship NRF-NRFI06-2020-0001. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the views of National Research Foundation, Singapore and Cyber Security Agency of Singapore. Daoyuan Wu was also partially supported by an HKUST grant.

REFERENCES

[1] L. Whitney, "Google paid out $10 million in bug bounties to security researchers in 2023," https://www.zdnet.com/article/google-paid-out-10-million-in-bug-bounties-to-security-researchers-in-2023/, Mar. 2024.

[2] Z. Zhang, B. Zhang, W. Xu, and Z. Lin, "Demystifying Exploitable Bugs in Smart Contracts," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, May 2023, pp. 615–627.

[3] 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)*, May 2019, pp. 8–15.

[4] Y. Fang, D. Wu, X. Yi, S. Wang, Y. Chen, M. Chen, Y. Liu, and L. Jiang, "Beyond "protected" and "private": An empirical security analysis of custom function modifiers in smart contracts," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2023, New York, NY, USA, 2023, p. 1157–1168.

[5] 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*, ser. PLDI 2020, New York, NY, USA, 2020, p. 454–469.

[6] J. Chen, X. Xia, D. Lo, J. Grundy, X. Luo, and T. Chen, "Defectchecker: Automated smart contract defect detection by analyzing evm bytecode," *IEEE Transactions on Software Engineering*, vol. 48, no. 7, p. 2189–2207, Jul. 2022.

[7] 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," no. arXiv:1809.03981, Sep. 2018.

[8] S. Kalra, S. Goel, M. Dhawan, and S. Sharma, "ZEUS: Analyzing safety of smart contracts," in *Proc. ISOC NDSS*, 2018.

[9] P. Tsankov, A. Dan, D. Drachsler-Cohen, A. Gervais, F. Bünzli, and M. Vechev, "Securify: Practical security analysis of smart contracts," in *Proc. ACM CCS*, 2018.

[10] 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)*, Nov. 2019, p. 1186–1189.

[11] Defillama, "Defillama hacks," 2024. [Online]. Available: https://defillama.com/hacks

[12] I. David, L. Zhou, K. Qin, D. Song, L. Cavallaro, and A. Gervais, "Do you still need a manual smart contract audit?" no. arXiv:2306.12338, Jun. 2023.

[13] Y. Sun, D. Wu, Y. Xue, H. Liu, H. Wang, Z. Xu, X. Xie, and Y. Liu, "GPTScan: Detecting Logic Vulnerabilities in Smart Contracts by Combining GPT with Program Analysis," in *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, 2024.

[14] S. Hu, T. Huang, F. Ilhan, S. Tekin, and L. Liu, "Large language model-powered smart contract vulnerability detection: New perspectives," in *2023 5th IEEE International Conference on Trust, Privacy and Security in Intelligent Systems and Applications (TPS-ISA)*, Los Alamitos, CA, USA, nov 2023, pp. 297–306.

[15] Y. Sun, D. Wu, Y. Xue, H. Liu, W. Ma, L. Zhang, M. Shi, and Y. Liu, "LLM4Vuln: A Unified Evaluation Framework for Decoupling and Enhancing LLMs' Vulnerability Reasoning," no. arXiv:2401.16185, Jan. 2024.

[16] P. Lewis, E. Perez, A. Piktus, F. Petroni, V. Karpukhin, N. Goyal, H. Küttler, M. Lewis, W.-t. Yih, T. Rocktäschel, S. Riedel, and D. Kiela, "Retrieval-augmented generation for knowledge-intensive NLP tasks," in *Proceedings of the 34th International Conference on Neural Information Processing Systems*, ser. NIPS'20. Red Hook, NY, USA: Curran Associates Inc., Dec. 2020, pp. 9459–9474.

[17] K. Tian, E. Mitchell, H. Yao, C. D. Manning, and C. Finn, "Fine-tuning Language Models for Factuality," no. arXiv:2311.08401, Nov. 2023.

[18] K. Lv, Y. Yang, T. Liu, Q. Gao, Q. Guo, and X. Qiu, "Full Parameter Fine-tuning for Large Language Models with Limited Resources," no. arXiv:2306.09782, Jun. 2023.

[19] A. Balaguer, V. Benara, R. L. d. F. Cunha, R. d. M. E. Filho, T. Hendry, D. Holstein, J. Marsman, N. Mecklenburg, S. Malvar, L. O. Nunes, R. Padilha, M. Sharp, B. Silva, S. Sharma, V. Aski, and R. Chandra, "RAG vs Fine-tuning: Pipelines, Tradeoffs, and a Case Study on Agriculture," no. arXiv:2401.08406, Jan. 2024.

[20] iAudit, "iaudit inference code and dataset," 2024. [Online]. Available: https://sites.google.com/view/iauditttool/home

[21] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017.

[22] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," in *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, Minneapolis, Minnesota, Jun. 2019, p. 4171–4186.

[23] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou, "CodeBERT: A pre-trained model for programming and natural languages," in *Findings of the Association for Computational Linguistics: EMNLP 2020*, T. Cohn, Y. He, and Y. Liu, Eds., Online, Nov. 2020, pp. 1536–1547.

[24] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, "Language models are unsupervised multitask learners," 2019.

[25] 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*, vol. 33, pp. 1877–1901, 2020.

[26] M. Lewis, Y. Liu, N. Goyal, M. Ghazvininejad, A. Mohamed, O. Levy, V. Stoyanov, and L. Zettlemoyer, "Bart: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension," no. arXiv:1910.13461, Oct. 2019.

[27] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu, "Exploring the limits of transfer learning with a unified text-to-text transformer," *J. Mach. Learn. Res.*, vol. 21, no. 1, jan 2020.

[28] Y. Wang, W. Wang, S. Joty, and S. C. Hoi, "Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation," in *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, 2021, pp. 8696–8708.

[29] W. X. Zhao, K. Zhou, J. Li, T. Tang, X. Wang *et al.*, "A survey of large language models," no. arXiv:2303.18223, Nov. 2023.

[30] Y. Chang, X. Wang, J. Wang, Y. Wu, L. Yang, K. Zhu, H. Chen, X. Yi, C. Wang, Y. Wang, W. Ye, Y. Zhang, Y. Chang, P. S. Yu, Q. Yang, and X. Xie, "A survey on evaluation of large language models," *ACM Trans. Intell. Syst. Technol.*, Jan 2024, just Accepted.

[31] J. Kaplan, S. McCandlish, T. Henighan, T. B. Brown, B. Chess, R. Child, S. Gray, A. Radford, J. Wu, and D. Amodei, "Scaling laws for neural language models," no. arXiv:2001.08361, Jan. 2020.

[32] Google, "Google gemini ai," 2024. [Online]. Available: https://blog.google/technology/ai/google-gemini-ai

[33] H. Touvron, L. Martin, K. Stone *et al.*, "Llama 2: Open foundation and fine-tuned chat models," no. arXiv:2307.09288, Jul. 2023, arXiv:2307.09288 [cs].

[34] L. Xu, H. Xie, S.-Z. J. Qin, X. Tao, and F. L. Wang, "Parameter-efficient fine-tuning methods for pretrained language models: A critical review and assessment," no. arXiv:2312.12148, Dec. 2023.

[35] Z. Wan, X. Wang, C. Liu, S. Alam, Y. Zheng, J. Liu, Z. Qu, S. Yan, Y. Zhu, Q. Zhang, M. Chowdhury, and M. Zhang, "Efficient large language models: A survey," no. arXiv:2312.03863, Jan. 2024.

[36] Q. Dong, L. Li, D. Dai, C. Zheng, Z. Wu, B. Chang, X. Sun, J. Xu, L. Li, and Z. Sui, "A survey on in-context learning," no. arXiv:2301.00234, Jun. 2023.

[37] Z. Hu, L. Wang, Y. Lan, W. Xu, E.-P. Lim, L. Bing, X. Xu, S. Poria, and R. Lee, "LLM-adapters: An adapter family for parameter-efficient fine-tuning of large language models," in *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, Singapore, Dec. 2023, pp. 5254–5276.

[38] W. Song, Z. Li, L. Zhang, H. Zhao, and B. Du, "Sparse is enough in fine-tuning pre-trained large language model," no. arXiv:2312.11875, Dec. 2023.

[39] E. J. Hu, P. Wallis, Z. Allen-Zhu, Y. Li, S. Wang, L. Wang, W. Chen *et al.*, "Lora: Low-rank adaptation of large language models," in *International Conference on Learning Representations*, 2021.

[40] X. L. Li and P. Liang, "Prefix-tuning: Optimizing continuous prompts for generation," in *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, 2021, pp. 4582–4597.

[41] B. Lester, R. Al-Rfou, and N. Constant, "The power of scale for parameter-efficient prompt tuning," in *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, Online and Punta Cana, Dominican Republic, Nov. 2021, pp. 3045–3059.

[42] N. Mundra, S. Doddapaneni, R. Dabre, A. Kunchukuttan, R. Puduppully, and M. M. Khapra, "A comprehensive analysis of adapter efficiency," in *Proceedings of the 7th Joint International Conference on Data Science & Management of Data (11th ACM IKDD CODS and 29th COMAD)*, 2024, pp. 136–154.

[43] A. Petrov, P. Torr, and A. Bibi, "When do prompting and prefix-tuning work? a theory of capabilities and limitations," in *The Twelfth International Conference on Learning Representations*, 2024.

[44] D. A. Zetzsche, D. W. Arner, and R. P. Buckley, "Decentralized finance (defi)," *Journal of Financial Regulation*, vol. 6, pp. 172–203, 2020.

[45] Defillama, "Defillama chain," 2024. [Online]. Available: https://defillama.com/chains

[46] P. Praitheeshan, L. Pan, J. Yu, J. Liu, and R. Doss, "Security analysis methods on ethereum smart contract vulnerabilities: A survey," no. arXiv:1908.08605, Sep. 2020.

[47] P. Züst, T. Nadahalli, and Y. W. R. Wattenhofer, "Analyzing and preventing sandwich attacks in ethereum," *ETH Zürich*, 2021.

[48] C. Zhou, J. He, X. Ma, T. Berg-Kirkpatrick, and G. Neubig, "Prompt consistency for zero-shot task generalization," in *Findings of the Association for Computational Linguistics: EMNLP 2022*, Abu Dhabi, United Arab Emirates, Dec. 2022, pp. 2613–2626.

[49] B. Rozière, J. Gehring, F. Gloeckle, S. Sootla *et al.*, "Code llama: Open foundation models for code," no. arXiv:2308.12950, Jan. 2024.

[50] R. Taori, I. Gulrajani, T. Zhang, Y. Dubois, X. Li, C. Guestrin, P. Liang, and T. B. Hashimoto, "Stanford alpaca: An instruction-following llama model," https://github.com/tatsu-lab/stanford_alpaca, 2023.

[51] T. Kojima, S. S. Gu, M. Reid, Y. Matsuo, and Y. Iwasawa, "Large language models are zero-shot reasoners," *Advances in neural information processing systems*, vol. 35, pp. 22 199–22 213, 2022.

[52] A. Q. Jiang, A. Sablayrolles, A. Roux, A. Mensch, and et al., "Mixtral of experts," no. arXiv:2401.04088, Jan. 2024.

[53] A. Q. Jiang, A. Sablayrolles, A. Mensch, and et al., "Mistral 7b," no. arXiv:2310.06825, Oct. 2023.

[54] F. Xue, Z. Zheng, Y. Fu, J. Ni, Z. Zheng, W. Zhou, and Y. You, "Openmoe: An early effort on open mixture-of-experts language models," no. arXiv:2402.01739, Jan. 2024.

[55] Solodit, "Solodit," 2024. [Online]. Available: https://solodit.xyz/

[56] B. J. Frey and D. Dueck, "Clustering by passing messages between data points," *science*, vol. 315, no. 5814, pp. 972–976, 2007.

[57] X. Yi, D. Wu, L. Jiang, Y. Fang, K. Zhang, and W. Zhang, "An empirical study of blockchain system vulnerabilities: Modules, types, and patterns," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2022, pp. 709–721.

[58] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, L. Shujie, L. Zhou, N. Duan, A. Svyatkovskiy, S. Fu *et al.*, "Graphcodebert: Pre-training code representations with data flow," in *International Conference on Learning Representations*, 2020.

[59] D. Guo, S. Lu, N. Duan, Y. Wang, M. Zhou, and J. Yin, "Unixcoder: Unified cross-modal pre-training for code representation," in *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 2022, pp. 7212–7225.

[60] Huggingface, "Huggingface transformer," 2024. [Online]. Available: https://huggingface.co/docs/transformers/

[61] Y. Zhuang, Z. Liu, P. Qian, Q. Liu, X. Wang, and Q. He, "Smart Contract Vulnerability Detection using Graph Neural Network," in *Twenty-Ninth International Joint Conference on Artificial Intelligence*, vol. 3, Jul. 2020, pp. 3283–3290.

[62] 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," in *Twenty-Ninth International Joint Conference on Artificial Intelligence*, vol. 3, Aug. 2021, pp. 2751–2759.

[63] H. Wu, Z. Zhang, S. Wang, Y. Lei, B. Lin, Y. Qin, H. Zhang, and X. Mao, "Peculiar: Smart Contract Vulnerability Detection Based on Crucial Data Flow Graph and Pre-training Techniques," in *2021 IEEE 32nd International Symposium on Software Reliability Engineering (ISSRE)*, Oct. 2021, pp. 378–389.

[64] S. Ullah, M. Han, S. Pujar, H. Pearce, A. Coskun, and G. Stringhini, "Llms cannot reliably identify and reason about security vulnerabilities (yet?): A comprehensive evaluation, framework, and benchmarks," in *2024 IEEE Symposium on Security and Privacy (SP)*, Los Alamitos, CA, USA, may 2024, pp. 199–199.

[65] M. Fu, C. Tantithamthavorn, V. Nguyen, and T. Le, "Chatgpt for vulnerability detection, classification, and repair: How far are we?" in *2023 30th Asia-Pacific Software Engineering Conference (APSEC)*, Los Alamitos, CA, USA, dec 2023, pp. 632–636.

[66] 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*, ser. ACSAC '22. New York, NY, USA: Association for Computing Machinery, Dec. 2022, pp. 481–496.

[67] M. Alqarni and A. Azim, "Low Level Source Code Vulnerability Detection Using Advanced BERT Language Model," in *Proceedings of the Canadian Conference on Artificial Intelligence*. Canadian Artificial Intelligence Association (CAIAC), May 2022.

[68] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding," in *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, J. Burstein, C. Doran, and T. Solorio, Eds. Minneapolis, Minnesota: Association for Computational Linguistics, Jun. 2019, pp. 4171–4186.

[69] H. Li, Y. Hao, Y. Zhai, and Z. Qian, "Assisting static analysis with large language models: A chatgpt experiment," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2023, New York, NY, USA, 2023, p. 2107–2111.

[70] S. J. Wang, K. Pei, and J. Yang, "Smartinv: Multimodal learning for smart contract invariant inference," in *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, 2024, pp. 126–126.

[71] Y. Liu, Y. Xue, D. Wu, Y. Sun, Y. Li, M. Shi, and Y. Liu, "Propertygpt: Llm-driven formal verification of smart contracts through retrieval-augmented property generation," *arXiv preprint arXiv:2405.02580*, 2024.

[72] G. Team, T. Mesnard, C. Hardin, R. Dadashi, S. Bhupatiraju, S. Pathak *et al.*, "Gemma: Open models based on gemini research and technology," no. arXiv:2403.08295, Mar. 2024.