Leveraging Fine-Tuned Language Models for Efficient and Accurate Smart Contract Auditing

ZHIYUAN WEI, Beijing Institute of Technology, China

JING SUN, University of Auckland, New Zealand

ZIJIAN ZHANG, Beijing Institute of Technology, China

XIANHAO ZHANG, Beijing Institute of Technology, China

MENG LI, Hefei University Of Technology, China

The rise of blockchain technologies has greatly accelerated the development and deployment of smart contracts. However, their inherent vulnerabilities and susceptibility to bugs have led to significant financial losses, underscoring the challenges in securing smart contracts. While traditional auditing methods are crucial, they often fall short in addressing the increasing complexity and volume of smart contracts. Recent advancements in Large Language Models (LLMs) offer promising solutions for enhancing software auditing by automatically identifying security vulnerabilities. Despite their potential, the practical application of these models is hindered by substantial computational demands. This paper investigates the feasibility of using smaller, fine-tuned models to achieve comparable or even superior results in smart contract auditing. We introduce the FTSmartAudit framework, which is designed to develop cost-effective, specialized models for smart contract auditing through the fine-tuning of LLMs. Our contributions include: (1) a single-task learning framework that streamlines data preparation, training, evaluation, and continuous learning; (2) a robust dataset generation method utilizing domain-special knowledge distillation to produce high-quality datasets from advanced models like GPT-4o; (3) an adaptive learning strategy to maintain model accuracy and robustness; (4) the proven effectiveness of fine-tuned models in detecting specific vulnerabilities and complex logical errors; and (5) a framework that can be extended to other domains requiring LLM solutions. Our experimental results demonstrate that smaller models can surpass state-of-the-art commercial models and tools in detecting vulnerabilities in smart contracts.

1 INTRODUCTION

Recent advancements in distributed ledger technologies have precipitated significant progress in the domain of smart contracts, with notable implications for applications such as cybersecurity and decentralized finance. Smart contracts, defined as self-executing programmatic agreements with predefined conditions established by participating entities, have emerged as pivotal innovations in decentralized systems. While smart contracts offer significant advantages in

terms of automation and trustless transactions, they are also prone to vulnerabilities and bugs that can lead to severe financial and operational consequences. Over the past half-decade, the ecosystem has witnessed a series of high-profile security breaches and exploitations targeting smart contracts, such as the DAO attack [8]. Consequently, ensuring the security of smart contracts remains a complex challenge, and traditional auditing methods, while essential, are increasingly inadequate in addressing the growing complexity and scale of these contracts.

Large Language Models (LLMs), including BERT [13], T5 [26], and GPT [9], have demonstrated potential in automating vulnerability detection by extracting key features and providing accurate predictions [20, 26, 40]. For example, Zhang et al. [39] highlighted GPT's success in detecting vulnerabilities in C/C++ and Java, often outperforming traditional methods. However, despite the impressive capabilities of these state-of-the-art (SOTA) commercial LLMs, their deployment in real-world applications is challenging due to their high computational requirements. Serving a single LLM with 175 billion parameters requires at least 350 GB of GPU memory, using specialized infrastructure [44]. To make matters worse, many researchers believe that larger models tend to achieve better performance, which has led to the development of models exceeding 500 billion parameters. Such computational requirements are far beyond the reach of most users, particularly for applications that require low-latency performance. Additionally, large models may contain excess data, which can degrade performance in some domain-specific tasks [17, 43]. Significantly, models with large parameters are primarily intended for multitasking. For just one or two specific tasks, using these extensive models is generally excessive. Consequently, it is essential to pursue alternative strategies that enhance the feasibility and efficiency of deploying LLMs.

While large models are powerful multitaskers, they are often excessive for domain-specific tasks, such as smart contract auditing. In contrast, smaller models, which are more computationally efficient, present a promising alternative. A major limitation in smart contract auditing is the lack of specialized, high-quality datasets. Existing tools and models may detect general vulnerabilities, but they often struggle with the nuances of smart contracts, where even minor logic flaws can result in substantial financial losses. For smaller models to maintain high accuracy while reducing computational overhead, they require focused, high-quality datasets. However, the scarcity of such datasets hampers their effectiveness. This leads to the central question of our research: **How can we enable smaller models to efficiently learn and detect vulnerabilities using compact, high-quality datasets in smart contract auditing?**

To address this challenge, we introduce the FTSmartAudit framework[1], a novel approach that combines high-quality dataset extraction from raw data with adaptive learning to continuously enhance both model performance and dataset quality. Our framework is specifically designed to train smaller models for efficient and accurate vulnerability detection in smart contracts. The key contributions of our work include:

- **Automatic Framework**: We propose a single-task learning framework tailored for smart contract auditing. This framework integrates four stages of creating specialized models, including data preparation, training process, evaluation, and continuous learning. It helps researchers and practitioners focus on refining model accuracy and performance.

- **Domain-Special Knowledge Distillation**: We provide a high-quality dataset generation method tailored for smart contract auditing. This method leverages domain-special knowledge distillation by using frontier models like GPT-4o to generate data, which is then used to fine-tune and improve the performance of more cost-efficient models. We ensure the dataset covers a comprehensive range of vulnerability types and is representative of many real-world scenarios.

---

[1]We have released all artifacts of our tool at https://github.com/LLMSmartAudit/FTSmartAudit

- **Adapted Learning**: We introduce a novel adapted training framework that improves data and model based on iterative feedback from each training round. This approach addresses the challenge of integrating new knowledge without introducing noise, which can degrade model performance. By continuously updating the model with new high-quality data while maintaining its core capabilities, we ensure the model remains robust and accurate over time.
- **Vulnerability Detection**: Our study shows that with effective fine-tuning, the fine-tuned models not only excel in detecting specific types of vulnerabilities but also improve in comprehending complex logical errors and subtle security flaws. These capabilities enhance the overall security and reliability of smart contracts.
- **Model Extensibility**: Although primarily designed for smart contract auditing, our framework is inherently flexible and can be adapted to other domains that require LLM solutions. This versatility allows the framework to address a wide range of tasks, including code analysis, regulatory compliance checks, and other security-related applications.

Our experimental results confirm the effectiveness of the FTSmartAudit framework. The method converges stably during training and significantly improves detection accuracy compared to baseline models. In zero-day tests, it successfully identified previously undetected logical vulnerabilities, underscoring its robustness and practical value in real-world smart contract auditing.

The rest of the paper is organized as follows. Section 2 presents the current issues in smart contract analysis and the use of LLMs for vulnerability prediction. Section 3 details the design and operational mechanism of FTSmartAudit. Section 4 presents the process of fine-tuning LLMs for the specialized task. In Section 5, we conduct a comprehensive evaluation of our system with three validation datasets, experimental criteria, and experimental results. Section 6 summarizes the works related to LLMs for vulnerability prediction, discusses our findings, and addresses potential threats to validity. Finally, Section 7 concludes the paper by summarizing our contributions and offering future insights.

2 BACKGROUNDS

2.1 Smart Contract Security

Smart contracts have emerged as a transformative force in the digital realm, giving rise to a wide range of compelling applications. Recent surveys [37, 46] indicate a rapid increase in the number of smart contracts over the past five years. DeFi, the most important application of smart contracts, has seen a significant rise in popularity, with its peak total value locked (TVL) reaching 179 billion in USD on 9 November 2021 [2]. However, the substantial asset values associated with smart contracts also attract numerous potential malicious actors. Smart contracts have been plagued by several high-profile vulnerabilities and exploits. Zhou et al. [46] document that smart contracts have suffered from countless high-profile attacks, resulting in losses exceeding 3.24 billion in USD from April 2018 to April 2022.

Unlike traditional software, smart contracts are more prone to having vulnerabilities permanently embedded within their code. Thus, many security researchers try to find analysis tools to automatically detect and analyze smart contracts before they are deployed. They employ advanced techniques such as formal verification, symbolic execution, fuzzing, and intermediate representation (IR) to enhance their effectiveness [8, 34]. While essential, these traditional auditing methods are often time-consuming and may not scale well with the growing complexity and number of smart contracts. For instance, tools relying on formal verification are adept at ensuring contracts adhere to specified requirements but may fall short in detecting security flaws like reentrancy or gas limit issues. Similarly, fuzzing is more effective in finding shallow bugs and less effective in identifying bugs that lie deep in the execution flow.

---

[2]https://defillama.com/

2.2   LLMs for Vulnerability Prediction

LLMs are termed "large" due to their extensive number of parameters, which empower them to comprehend and generate human language with remarkable coherence and contextual appropriateness. Natural language processing, image generation, code, and mathematical problem-solving are on the topic of target domains. In the realm of code processing, LLMs have shown considerable advancement since the pioneering work of Codex [9]. This progress has led to the development of commercial products like GitHub Copilot [4] and open-source code models such as StarCoder [21] and Codellama [28].

LLMs have also achieved excellent performance on specific downstream tasks, such as code analysis, vulnerability detection and code upgrading [14]. Chen et al. [10] have proven that LLMs (GPT-2, T5), trained with a high-quality dataset consisting of 18,945 vulnerable functions about C/C++, outperform other machine learning methods, such as Graph Neural Networks, in vulnerability detection. Additionally, fine-tuned models such as CodeT5 and NatGen significantly improve the performance on vulnerability detection task. SOTA commercial products, such as GPT-4, offer promising solutions to augment the smart contract auditing process [7, 11]. By leveraging their code comprehension and generation capabilities, these models can identify specific vulnerabilities, verify compliance, and check for logical correctness. Their effectiveness is further enhanced through advanced prompting techniques like chain-of-thought (CoT) or few-shot.

2.3   Fine-tuning Technique

Fine-tuning is a technique that adapts a pre-trained model's general knowledge to perform well on specific domains or tasks by further training the model on a targeted dataset [25]. This approach allows the model to learn task-specific features while retaining its broad language understanding. The effectiveness of fine-tuning depends on the base model's exposure to similar data. If the model has never encountered the type of data you're interested in, fine-tuning may have limited impact. However, when applied correctly, fine-tuning can yield impressive results even with relatively small datasets.

Compared to retraining all of a model's parameters from scratch, fine-tuning is often more efficient and can achieve significant improvements with fewer resources. For example, CodeLlama model [28] was created by fine-tuning Llama 2 on a mix of proprietary instruction data. This process significantly improved performance on various truthfulness, toxicity, and bias benchmarks while maintaining strong code generation capabilities. CodeGemma [33] also illustrates the power of domain-specific fine-tuning. By training on more than 500 billion tokens of code, CodeGemma achieved significantly better performance on coding tasks compared to the base Gemma models. Furthermore, Lima et al. [45] demonstrated that fine-tuning a 65 billion parameter LLaMA (v1) model on just 1,000 high-quality samples could outperform GPT-3 (DaVinci003) on certain tasks.

3   METHODOLOGY

We have developed a framework named FTSmartAudit (Fine-Tuning for Smart Contract Audit), which utilizes LLMs for smart contract auditing. This framework has proven effective across various model sizes and architectures. As illustrated in Figure 1, FTSmartAudit is structured around four main stages: i.e., data preparation, training, evaluation, and continuous learning. These stages create a continuous cycle that facilitates ongoing enhancements in model performance.
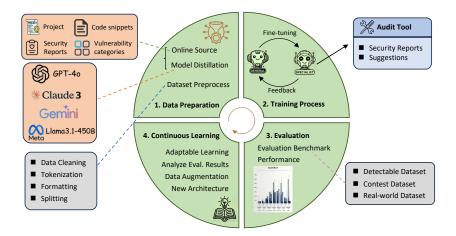
Fig. 1. Overview of FTSmartAudit framework

In the data preparation stage, detailed in Section 3.1, we collect and process the necessary materials to train the model. The training process, described in Section 3.2, involves fine-tuning the selected LLM with the prepared data to optimize its performance for smart contract auditing tasks. Following this, as presented in Section 5, we conduct a comprehensive evaluation to determine if the model meets our performance expectations. Based on the evaluation results, we adopt a continuous learning approach, outlined in Section 3.3, aimed at enhancing both the model's performance and the quality of our fine-tuning data. This stage includes analyzing strengths and weaknesses, identifying improvement areas, and refining our strategies. Upon completing the continuous learning stage, we circle back to the data preparation step, initiating a new cycle that ensures the continuous refinement and enhancement of the FTSmartAudit framework.

3.1 Data Preparation

High-quality fine-tuning data is crucial for developing effective models. Fine-tuning datasets offer extensive text that enables models to learn grammar, syntax, and the overall structure of the language. Although not flawless, fine-tuning equips models with a basic level of common-sense reasoning through exposure to patterns and relationships in the data. This process aids in understanding cause and effect, making inferences, and recognizing typical event sequences. By fine-tuning on large datasets, the model learns representations that can be fine-tuned for specific tasks. This fine-tuning lays a robust foundation, allowing the model to be further adapted to specific applications through additional task-specific training. To support this, we have developed a data generation approach to create a high-quality dataset of smart contract code samples, which consists of two primary steps: data collection and dataset construction.

3.1.1 Data Collection. Data collection for the fine-tuning dataset is derived from two primary sources: manually labeled data and automatically generated data.

Manually Labeled Data: This category of data encompasses two specific types: open-source codebases and security reports. The open-source codebase includes publicly accessible smart contract code from various repositories, which are meticulously reviewed and annotated by security experts. The security reports are detailed documents prepared by security auditors or security firms, documenting specific vulnerabilities, attacks, and fixes in smart contracts. The
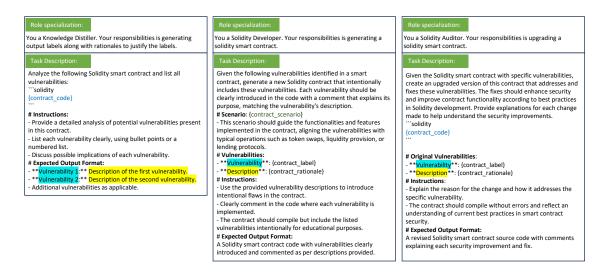
Fig. 2. The instructional templates of LLM agents: the left one is the Distillation Agent, the middle one is the Developer Agent, and the right one is the Security Agent. Each agent is constructed by prompting. Prompting includes both an example rationale (highlighted in yellow) and a label (highlighted in blue).

integrity and relevance of these sources are ensured as they have been verified and labeled by experts. The main sources include:

- Decentralized Application Security Project (DASP) [3]: Focuses on the security of decentralized applications, especially on blockchain platforms like Ethereum. DASP's Top 10 list underscores the most critical security vulnerabilities found in smart contracts.
- Smart Contract Weakness Classification (SWC) [4]: Provides a structured classification of security vulnerabilities specific to Ethereum contracts, paralleling the Common Weakness Enumeration system. This helps in systematically addressing and understanding smart contract weaknesses.
- Slither detectors [5]: A static analysis framework for Solidity, containing various functions for smart contract analysis. It also provides reports and detailed visual insights about vulnerabilities, covering up to 94 different cases.
- DeFiVulnLabs [6]: explore DeFi protocols to uncover common and novel vulnerabilities in smart contracts. Now, it supports 47 types of vulnerabilities. Besides, it could produce educational content, such as tutorials and guides.
- Code4rena [7]: Operates as a competitive auditing platform that fosters smart contract security through community-driven audits. Experts and companies globally identify vulnerabilities in real-world smart contract projects, motivated by bounties to uncover severe vulnerabilities.

Synthetic Data: We employ knowledge distillation, also known as model distillation, to generate synthetic examples of code and analysis reports. Knowledge distillation is a technique where domain-specific knowledge from large parameter models is distilled into smaller models, enhancing their domain-specific task-solving capabilities and reducing inference latency [5, 16]. This approach is particularly useful in scenarios with limited labeled data, as it

---

[3]ttps://dasp.co/

[4]https://swcregistry.io/

[5]https://github.com/crytic/slither

[6]ttps://github.com/SunWeb3Sec/DeFiHackLabs

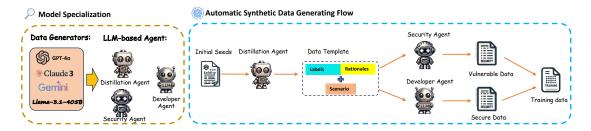[7]https://github.com/ZhangZhuoSJTU/ Web3Bugs

Fig. 3. The workflow of knowledge distillation

leverages a larger "teacher" model to generate a training dataset with noisy pseudo labels [3], [18]. We adopt this method to create a synthetic dataset from large parameter models.

Large parameter models (such as GPT and Claude) have proven effective in generating synthetic data to train smaller models [18, 35]. Although data labeled by these large parameter models tends to be noisier than manually labeled data, the process offers significant advantages in cost-effectiveness, speed, and generalizability across various tasks. To optimize and automate our synthetic data generation porcess, we implemented LLM agent techniques [19, 42], creating agents four times, each time using one of the following models: GPT-4o, Claude3, Gemini1.5, and Llama3.1-405B. We created three specialized agents: the Distillation Agent, the Solidity Developer Agent, and the Security Agent. Detailed prompt templates for these agents are presented in Figure 2. The roles of these agents are described as follows:

- *Distillation Agent*: Generates output labels with rationales that justify these labels, focusing on identifying the type of vulnerability and providing explanations.
- *Developer Agent*: Creates smart contract code that includes specific vulnerabilities.
- *Security Agent*: Creates secure smart contract to address previously identified vulnerabilities. Secure contracts that are similar to vulnerable ones, but with subtle differences.

The workflow knowledge distillation can be divided into two parts: model specialization and adaptive data generation, as illustrated in Figure 3. Model specialization tailors the target models to focus on specific tasks. Adaptive data generation not only streamlines the workflow but also ensures continuous improvement.

We formalize the process as follows: Let $\mathcal{D} = \{x_i\}_{i=1}^N$ denote a dataset where each x_i is a contract code, y_p a label, r_p a rationale, and s_p a scenario containing at least one vulnerability. Given an initial seed $x_i \in \mathcal{D}$, the Distillation Agent $\mathcal{A}_{Di}$ first generates a JSON report with the corresponding label y_p with rationale r_p. A scenario s_p is selected from a predefined set of real-world application relevant to the smart contract domain. Next, we create a triplet (y_p, r_p, s_p) and pass it to the Developer Agent $\mathcal{A}_{de}$, which generates new vulnerable data x_v. The contract code x_v, the target label y_p, and the updated rationales $\widehat{r}_p$ are added as a new entry in the vulnerable dataset $\mathcal{S}_v$. This vulnerable data x_v is then passed to the Security Agent $\mathcal{A}_{ds}$, which transforms the vulnerable contract code into secure contract code x_c with the corresponding security label y_c. Finally, the vulnerable contract report and the secure contract report are combined into a synthetic training dataset $\mathcal{S}_{v,c}$. This entire workflow is formalized in Algorithm 1.

In this study, the Developer Agent does not directly receive input labels for a specific reason: the target LLM operates as a black-box system, which restricts direct access to the complete list of vulnerability labels. Instead, we initiate the process by selecting various smart contracts as initial seeds in an effort to exhaustively cover the spectrum of potential vulnerabilities. This approach aims to push the boundaries of the model's capability in identifying diverse types of security flaws. Finally, we obtain a raw fine-tuning dataset containing more 2,000 entries.

Algorithm 1 Knowledge Distillation

1: **Input:** Initial seed contract code $x_i \in \mathcal{D}$ with vulnerability
2: **Output:** Synthetic training dataset $\mathcal{S}_{v,c}$
3: Initialize Agents $\mathcal{A}_{Di}$, $\mathcal{A}_{de}$, and $\mathcal{A}_{ds}$
4: Generate JSON report by $\mathcal{A}_{Di}$ using x_i
5:      **Output**: Label y_p with rationale r_p
6: Define scenario s_p based on a set of real-world applications
7:      s_p: Select from a predefined set of real-world application scenarios relevant to the smart contract domain.
8: Create a triplet (y_p, r_p, s_p)
9: Pass triplet (y_p, r_p, s_p) to $\mathcal{A}_{de}$
10:      $\mathcal{A}_{de}$ generates new vulnerable data x_p
11:      Add contract code x_v, target label y_p, and updated rationale $\widehat{r}_p$ as a new entry to the vulnerable dataset $\mathcal{S}_v$
12: Pass vulnerable data x_p to $\mathcal{A}_{ds}$
13:      $\mathcal{A}_{ds}$ updates contract code x_v to secure contract x_c
14:      Add secure contract code x_c and security label y_c as a new entry to the secure dataset $\mathcal{S}_c$
15: Combine vulnerable dataset $\mathcal{S}_v$ and secure dataset $\mathcal{S}_c$
16:      **Output**: Combined synthetic dataset $\mathcal{S}_{v,c}$

3.1.2 Dataset Preprocessing. The 2000 entries in the raw fine-tuning dataset are usually not immediately suitable for training. This is because data from different sources may have varying formats, structures, or encodings. Some content may be of low quality, contain errors, or be inappropriate. Raw datasets frequently contain duplicates that can bias the model. To address these issues, several preprocessing steps are typically performed, including data cleaning, formatting, content filtering, and de-duplication. Since raw datasets collected from various sources (e.g., web scrapes) may contain useless comments, non-textual elements, excessive white-space, and inconsistent line breaks, these elements can introduce noise and hinder effective learning. Human involvement may be necessary to clean this data, ensuring that only relevant and high-quality entry is retained.

Each model is configured to operate with a specific template. Although pre-trained models like Llama2 excel at predicting the next token in a sequence, they require explicit instruction to generate useful responses. Prior literatures have shown that increasing the number of tasks in fine-tuning with instructions improves generalization to unseen tasks [29, 36]. Inspired by the Evol-Instruct method proposed in WizardLM's [38], we designed a data template to enhance instruction fine-tuning effectiveness (Figure 4). This template contains three components: instruction, input, and output, enhancing the model's ability to follow and execute commands.

The maximum context size is another constraint for model training. For instance, the Llama 2 model can only process up to 4,096 tokens. During preparation, we tokenize the dataset's content into manageable units and remove data exceeding this limit (4,096 tokens) to prevent input overflow. Additionally, redundant data, especially prevalent in smart contracts, can prolong training times and degrade model performance. To mitigate this issue, we employ the sentence embeddings model *paraphrase-MiniLM-L6-v2* [27] for deduplication, which is widely applied in NLP tasks to find semantically similar sentences in large corpora. Setting a similarity threshold close to 0.9 allows us to maintain high similarity between inputs, effectively removing duplicates while preserving unique information.

3.2 Training Process

In this study, we employed two primary techniques for fine-tuning models: SFT and QLoRA.

3.2.1 Supervised Fine-Tuning (SFT). SFT involves training models on a curated dataset consisting of instructions paired with appropriate responses. Unlike pre-training, which relies on vast amounts of general text data, SFT uses smaller, high-quality datasets specific to the desired task. During SFT, the model's weights are adjusted to minimize discrepancies
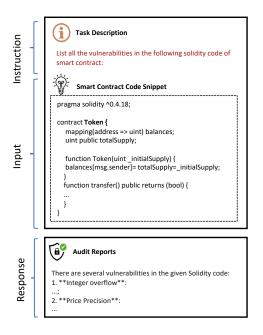
Fig. 4. Instruction Format Input Entry for LLMs

between its outputs and the provided ground-truth responses, enhancing the model's ability to generate expected answers based on the training entries.

SFT builds directly on the pretrained knowledge base, making it suitable for scenarios where specific outputs are desired. This technique fine-tunes the model's understanding by making small, targeted adjustments to its weights, refining its responses based on specific instructions or data encountered during the fine-tuning phase.

3.2.2 Quantized Low-Rank Adaptation (QLoRA). We also implemented QLoRA for its efficiency and cost-effectiveness. QLoRA is a high-fidelity 4-bit fine-tuning method that drastically reduces memory usage, enabling the fine-tuning of a 65-billion parameter model on a single 48GB GPU. This approach conserves resources while retaining performance levels comparable to those of full fine-tuning and Low-Rank Adaptation (LoRA) [6, 12].

Practitioners typically choose from three strategies for fine-tuning: full fine-tuning, LoRA, and QLoRA. Full fine-tuning involves retraining all or most of the model's parameters on the new dataset, while LoRA reduces complexity by altering a smaller, specific subset of parameters. QLoRA combines quantization and low-rank adaptation for efficient and effective model tuning. We selected QLoRA for its ability to maintain high performance while significantly reducing computational requirements. These advancements are crucial for deploying SOTA models in resource-constrained environments, although users should consider potential trade-offs in speed or adaptability depending on the specific application.

By integrating SFT and QLoRA into our methodology, we aimed to optimize the fine-tuning process, ensuring both high-quality performance and resource efficiency. This combination allows us to refine the model's responses based on specific instructions or data encountered during the fine-tuning phase while conserving computational resources.
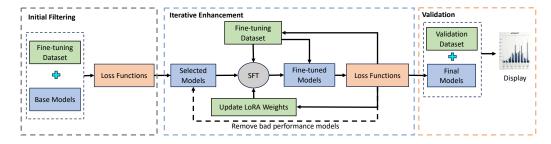
Fig. 5. Flow chart of FTSmartAudit's data and model enhancement

3.3 Continuous Learning

Introducing new knowledge to LLMs often introduces noise, which can reduce the accuracy of fine-tuned models. To address this, we have implemented a loss function method and an adaptive learning approach to enhance the quality of fine-tuned model.

3.3.1 Supervised Loss Functions. As a single-task learning framework, FTSmartAudit faces a significant challenge of fine-tuning data quality, the fine-tuned model performance. Inspired by MFTCoder [22], we incorporate a new approach specifically designed to evaluate the fine-tuned models. This approach combines label prediction accuracy with a rationale-based explanation system. To avoid the model favoring labels with larger amounts of data, we introduce a weight assignment strategy during loss computation. This strategy supports two calculation schemes: one based on the number of labels and the other based on the number of valid rationale involved in the loss calculation.

Formally, we denote the fine-tuning dataset as $\mathcal{D} = \{x_i, y_i, r_i\}_{i=1}^N$, where each x_i represents an input and y_i is the corresponding target label with the associated rationale r_i. The fine-tuned model f is trained to minimize the label prediction loss with cross-entropy loss for categorical outputs.

$$\mathcal{L}_{label} = -\frac{1}{N} \sum_{i=1}^{N} \log p(y_i|f(x_i)) \tag{1}$$

where $p(y_i|f(x_i))$ is the probability of the model outputting the correct label given the input x_i.

When computing rationale prediction loss, we employ a binary cross-entropy loss if rationales are binary (present/not present). The rationale prediction loss is computed as follows:

$$\mathcal{L}_{rationale} = -\frac{1}{N} \sum_{i=1}^{N} [r_i \log g(x_i) + (1 - r_i) \log(1 - g(x_i))] \tag{2}$$

where r_i represents the true label indicating the presence or absence of a rationale for the ith instance, and $g(x_i)$ is the model's prediction for rationale presence.

We integrate the label prediction loss and the rationale prediction loss using a dynamic weighting scheme based on the model's confidence in its predictions. The equation for the dynamic weighting scheme is as follows:

$$\mathcal{L} = \mathcal{L}_{label} + \lambda \mathcal{L}_{rationale} \tag{3}$$

where λ is a hyperparameter that controls the relative importance of the two losses.

3.3.2 Model and Data Quality Enhancement. By incorporating these different loss functions, FTSmartAudit effectively addresses the quality of the data and the performance of the models involved, as illustrated in Figure 5. The process is organized into three primary patterns: initial filtering, iterative process enhancement, and validation.

Formally, we denote the initial fine-tuning dataset as $\mathcal{D}(0) = \{x_i, y_i, r_i\}_{i=1}^{N}$ where x_i represents an input and y_i is the corresponding desired output label with the corresponding rationale r_i. Besides, the primary pre-trained model set is denoted as $\mathcal{M}_p = \{m_j\}_{j=1}^{M}$.

The first pattern of initial filtering involves selecting basic models from a variety of open-source LLMs. To broaden our scope of models, the label prediction loss, $\mathcal{L}_{label}$, serves as the criterion for filtering these models The threshold is set as $\mathcal{L}_b$. For each input x_i processed by model m_i, the output $\hat{y}i$ is obtained and $\mathcal{L}_{label}$ is computed. Models for which $\mathcal{L}_{label}$ falls below $\mathcal{L}_b$ are retained, forming a new model set $\mathcal{M}_s$.

The second pattern focuses on the iterative enhancement of model performance and data quality, as detailed in Algorithm 2. To refine our model selection future and enhance the fine-tuning dataset quality, we utilize a dual-threshold loss function, $\mathcal{L}$, characterized by two thresholds, $\mathcal{L}_l$ and $\mathcal{L}_h$ ($\mathcal{L}_l < \mathcal{L}_h$). The process is structured iteratively as follows:

1. *Initial Fine-Tuning*: The fine-tuning dataset $\mathcal{D}(0)$ is used to fine-tune the selected models $\mathcal{M}_s(0)$, resulting in the fine-tuned model set $\mathcal{M}_{ft}(0)$.

2. *Loss Evaluation and Actions:* For each model m_j in $\mathcal{M}_{ft}(0)$, outputs $\hat{y}_i$ and $\hat{r}_i$ are generated for each input x_i from $\mathcal{D}(0)$. After all inputs completed, the combined loss $\mathcal{L}_j$ for model m_j is computed using the values $\{\hat{y}_i, \hat{r}_i, y_i, r_i\}_{i=1}^{N}$. If $\mathcal{L}_j > \mathcal{L}_h$, the model m_j is deemed unsatisfactory and is removed from $\mathcal{M}_s$. If $\mathcal{L}_l \leq \mathcal{L}_j \leq \mathcal{L}_h$, the dataset $\mathcal{D}(k)$ is manually modified to improve its quality, resulting in $\mathcal{D}(k+1)$. If $\mathcal{L}_j < \mathcal{L}_l$, the dataset and model set are considered satisfactory, and the process transitions to the validation phase.

3. *Iterative Fine-Tuning*: The improved dataset $\mathcal{D}(1)$ is used for the next iteration of fine-tuning with the adjusted model set $\mathcal{M}_s(1)$, producing $\mathcal{M}_{ft}(1)$. The enhancement process similar to Step 2 is repeated.

4. *Cycle Continuation*: This iterative cycle is repeated until the model set $\mathcal{M}'_{ft}(k)$ and the dataset $\mathcal{D}(k)$ meet the desired standards. The cycle typically concludes successfully within a few iterations, as indicated by the condition $\mathcal{L}_j < \mathcal{L}_l$ being met or a set maximum number of iterations being reached.

Finally, the third pattern is validation, during which the satisfactory model set $\mathcal{M}'_{ft}(k)$ is subjected to further testing using a separate validation dataset, $\mathcal{D}_{valid}$. This final step confirms the robustness and effectiveness of the models in realistic settings.

The detailed iterative process of enhancing model and data quality is presented in Section A of the appendix.

In this section, we have outlined our framework, detailing the processes of data preparation, model training, and continuous learning. This foundation is essential for our study. The following section will elaborate on how we apply fine-tuning techniques to develop our specific models.

4 TASK-SPECIFIC FINE-TUNING

This section explores the intricate process of fine-tuning LLMs for the specific task of smart contract auditing.

4.1 Fine-tuning Dataset

The fine-tuning dataset provides a strong foundation for the model by exposing it to a wide variety of contract types and vulnerability instances. We create two fine-tuning datasets:

Algorithm 2 Iterative Enhancement Process

1: **Input:**
2:     Initial fine-tuning dataset $\mathcal{D}(0) = \{x_i, y_i, r_i\}_{i=1}^N$
3:     Primary pre-trained model set $\mathcal{M}_s(0) = \{m_j\}_{j=1}^M$
4:     Label prediction loss threshold $\mathcal{L}_b$
5:     Dual-thresholds for function loss $\mathcal{L}_l$ and $\mathcal{L}_h$ ($\mathcal{L}_l < \mathcal{L}_h$)
6: **Output:**
7:     Satisfactory fine-tuned model set $\mathcal{M}'_{ft}(K)$
8:     Satisfactory dataset $\mathcal{D}_{train} = \mathcal{D}(K)$
9: Initialize $k = 0$
10: **repeat**
11:     Fine-tune models in $\mathcal{M}_s(k)$ using $\mathcal{D}(k)$ to get $\mathcal{M}_{ft}(k)$
12:     **for** each model m_j in $\mathcal{M}_s(k)$ **do**
13:         Compute combined losses $\mathcal{L}$ for all inputs in $\mathcal{D}(k)$:
14:         **for** each input x_i in $\mathcal{D}(k)$ **do**
15:             Model m_i outputs $\hat{y}_i$ and $\hat{r}_i$ using x_i
16:         **end for**
17:         Compute combined loss $\mathcal{L}_j$ using $\{\hat{y}_i, \hat{r}_i, y_i, r_i\}_{i=1}^N$
18:         **if** $\mathcal{L}_j > \mathcal{L}_h$ **then**
19:             Remove m_j from $\mathcal{M}_s$
20:         **else if** $\mathcal{L}_l \leq \mathcal{L}_j \leq \mathcal{L}_h$ **then**
21:             Manually modify $\mathcal{D}(k)$ to form $\mathcal{D}(k+1)$
22:         **end if**
23:         **if** $\mathcal{L}_j < \mathcal{L}_l$ **then**
24:             Set $\mathcal{D}_{train} = \mathcal{D}(k)$
25:             **break**
26:         **end if**
27:     **end for**
28:     Increment k
29:     Update $\mathcal{M}_s(k+1) = \mathcal{M}_{ft}(k)$
30: **until** $\mathcal{L} < \mathcal{L}_l$ **or** condition for maximum iterations is met

---

- **FTAudit-Vuln Dataset**: This dataset comprises 975 vulnerable smart contracts, encompassing 120 distinct types of vulnerabilities (referred to as 'labels' in our study). It serves as a comprehensive collection of various smart contract vulnerabilities.

- **FTAudit Dataset**: This dataset is an extension of the FTAudit-Vuln dataset. It includes all 973 vulnerable contracts from FTAudit-Vuln and adds 850 secure Solidity contracts, bringing the total to 1,825 contracts. Secure contracts are included that are similar to vulnerable ones, but with subtle differences. This balanced dataset allows for more robust training and evaluation of vulnerability detection models.

Notably, in both dataset, we ensure that each vulnerability type has at least 5 code snippets with corresponding rationales. The distribution of the 120 vulnerability types are not evenly distributed. Some vulnerability types are more prevalent than others. Certain vulnerability types such as 'unchecked return calls', 'arithmetic bugs', and 'denial-of-service' are more common, as these areas have been verified by prior experiments where some models have shown weaknesses.

4.2 Model Adaptation

With many LLM models available, it is crucial to select the most suitable ones for our study. We have defined specific criteria for choosing models tailored to smart contract auditing:

- *Solidity Comprehension*: Ability to parse and comprehend Solidity, the primary language for Ethereum contracts.
- *Code-related Task Performance*: Proficiency in tasks such as code completion, vulnerability detection, and code summarization.
- *Semantic Understanding*: Capacity to grasp the semantic meaning of code beyond mere syntax.

- *Model Type*: Focus on completion models such as GPT and LLaMA series, which offering prediction for the most likely next words or tokens.
- *Model Size*: Preference for models under 10B parameters to demonstrate the efficacy of smaller models and reduce computational requirements.
- *Performance and Flexibility*: Emphasis on models that offer high performance and can be fine-tuned for specific smart contract auditing tasks.
- *Community and Support*: Prioritizing models with active community support and comprehensive documentation (e.g., those available on Hugging Face) for easier integration and troubleshooting.

Based on these criteria and model evaluation in Section A.1, we selected the following base models for our study:

- **Llama-3-8B-it**: Latest iteration of the LLaMA series, known for its efficiency and strong performance.
- **Gemma-7B-it**: Google's recent contribution, designed for responsible AI development.
- **CodeGemma-7B-it**: A code-specific variant of Gemma, potentially more suited for smart contract tasks.
- **Mistral-7B-it-v0.3**: Known for its MoE structure and strong performance despite its relatively small size.

The selected models use similar model size. We fine-tuned these models on FTAudit dataset and FTAudit-Vuln dataset. Our fine-tuned models[8] are shown in Table 1.

Table 1. FTAudit Model Series and Their Variants

Model Series	Models
FTAudit-Vuln-*	FTAudit-Vuln-Llama3-8b
	FTAudit-Vuln-Gemma-7B
	FTAudit-Vuln-CodeGemma-7B
	FTAudit-Vuln-Mistral-7B
FTAudit-*	FTAudit-Llama3-8b
	FTAudit-Gemma-7b
	FTAudit-CodeGemma-7b
	FTAudit-Mistral-7b

4.3 Fine-tuning Process

We conducted our model training using NVIDIA A100 GPUs, each equipped with 48GB of VRAM. For optimization, we employed the Adam optimizer with a learning rate of 5e-4. This specific learning rate was selected after extensive experimentation, as it effectively balanced convergence speed and stability for our particular task. To enhance efficiency and reduce memory requirements, we utilized the QLoRA fine-tuning method instead of LoRA. QLoRA quantizes and compresses the model's weights, allowing for effective fine-tuning with limited computational resources. This approach is particularly advantageous when working with LLMs on hardware with restricted memory capacity, as it significantly reduces the VRAM footprint without compromising the model's performance [47].

Additionally, the training process for all models was standardized at 1000 steps to ensure consistency across experiments. This number of steps was selected based on preliminary experiments which indicated it was sufficient for the models to reach a stable point in the loss landscape. To track model performance and stability, we use two metrics: training loss and grad norm. Figure 6 shows that the training loss consistently decreased to below 0.5 after approximately 800 iterations. Figure 7 shows that the gradient norm curve stabilized after approximately 600 steps. These two metrics reveal that our fine-tuned models had reached a stable point in the loss landscape, indicating effective learning.

---

[8]All models can be obtained from https://huggingface.co/weifar

In this section, we detail our application of fine-tuning techniques to develop our targeted fine-tuned models. The subsequent evaluation will assess the performance of our fine-tuned models against SOTA commercial products, providing a comprehensive comparison of the results.

Fig. 6. Training loss of fine-tuned models.

Fig. 7. The stability and dynamics of the training process.

5 EVALUATION

In this section, we will conduct multiple sets of experiments to validate the effectiveness and superiority of our fine-tuned models in the context of smart contract auditing.

5.1 Research Questions

Our evaluation aims to address the following research questions:

- **RQ1: How does fine-tuning base models impact their performance in detecting detectable vulnerabilities in smart contracts?** Given that many models, especially SOTA commercial products, have proven effective in identifying detectable vulnerabilities, it is essential to verify that our fine-tuned model possesses this basic capability. Additionally, we aim to assess whether the fine-tuning method enhances performance.
- **RQ2: How does the inclusion of secure smart contracts in the training dataset affect the performance of models in detecting vulnerabilities?** While vulnerable contracts are essential for expanding the models' knowledge

14

base, a dataset skewed heavily toward vulnerable contracts could lead to false positives. Measuring how the inclusion of secure contracts impacts metrics like accuracy and false positives would provide valuable insights.

- **RQ3: How do fine-tuned models perform in terms of detectable and undetectable bugs?** Complex logic bugs are considered some of the most severe security issues. We aim to evaluate the performance of LLMs in identifying both detectable and complex logic vulnerabilities, given the potential of LLMs to address these critical issues.
- **RQ4: How do fine-tuned models perform in detecting vulnerabilities in the large real-world dataset?** Analyzing the performance of fine-tuned models on large real-world contracts provides insights into their practical functionality. This helps identify potential gaps between theoretical capabilities and practical effectiveness.
- **RQ5: Can our fine-tuned models discover new vulnerabilities that were previously missed in the audit reports from Code4rena?** It is interesting to see if our fine-tuned models can find vulnerabilities that were missed in past Code4rena audit reports. This will help us understand if our models are more effective at identifying hidden issues.

By addressing these questions, our experiments aims to offer a comprehensive assessment of our fine-tuned models.

5.2 Evaluation Benchmark

To comprehensively measure the model's capabilities, we need robust validation datasets. As mentioned in [41], vulnerabilities in smart contracts can be categorized as either machine-auditable or machine-unauditable. Their survey indicates that existing tools can detect machine-auditable vulnerabilities, with more than 80% of exploitable bugs falling into this category. Some vulnerabilities are too complex or subtle and require the expertise of multiple human auditors. In our study, we refer to these machine-auditable vulnerabilities as "detectable vulnerabilities" and these machine-unauditable vulnerabilities as "undetectable vulnerabilities". We utilize three validation datasets: the Detectable dataset, the Contest dataset, and the Real-world dataset.

- **Detectable Dataset**: This dataset contains several detectable vulnerabilities that can be exploited by existing tools. We use the SmartBugs-curated dataset [1], which is widely used by developers and researchers for analyzing and improving the security of Ethereum smart contracts. It contains 143 annotated contracts with 180 tagged vulnerabilities, each marked with detectable vulnerabilities based on the DASP classification.
- **Contest Dataset**: Our Contest dataset is derived from the Code4rena contests [2], which encompass a collection of 102 projects and 6,454 contracts. From this extensive set, we selected 100 vulnerable smart contracts for our analysis. These contracts exhibited a total of 376 high and medium-risk vulnerabilities, as identified through the contest reports. Notably, 128 of these vulnerabilities correspond to the detectable vulnerabilities in the Detectable dataset.
- **Real-world Dataset**: This dataset [9] consists of actual contracts deployed on the Ethereum blockchain, ensuring that our evaluations reflect real-world usage patterns. We collected 11,510 contracts from Etherscan, spanning from July 8, 2017, to May 5, 2022. It is important to note that Etherscan does not store the source code for every contract, and some contracts with different addresses may contain identical content.

Additionally, we explored other datasets, such as those from DeFiVulnLabs [10] and DeFiHackLabs [11]. The DeFiVulnLabs dataset was used as training data for LLMs, with the GPT-4o model achieving a score of 37/50. We have included evaluation reports related to DeFiVulnLabs for these models in our repository. Conversely, DeFiHackLabs focuses on proof-of-concept DeFi hacking incidents, making it more suitable for attack analysis than vulnerability detection.

---

[9]https://huggingface.co/datasets/weifar/LargeRealworldDataset
[10]https://github.com/SunWeb3Sec/DeFiVulnLabs
[11]ttps://github.com/SunWeb3Sec/DeFiHackLabs

Table 2. Performance Comparison of Different Detection Tools on Labeled Dataset

Model	V1	V2	V3	V4	V5	V6	V7	V8	V9	Total	Recall
Slither-0.10.0	14	29	60	1	2	0	9	0	6	121	66.48%
Mythril-0.24.7	11	28	53	0	0	0	6	16	5	119	65.38%
Llama3-8b-it	12	30	35	1	4	0	16	17	3	118	64.84%
Gemma-7b-it	7	23	19	0	0	1	7	15	5	77	42.31%
Codegemma-7b-it	5	17	10	0	2	1	7	17	0	59	32.42%
Mistral-7b-it-v0.3	7	22	41	1	5	0	14	21	3	114	62.64%
GPT-4o	13	30	56	0	6	2	18	21	6	152	84.44%
Claude-3-Opus	13	30	49	0	6	2	18	21	6	145	80.56%
Gemini-1.5-pro	14	30	62	1	5	3	15	20	5	155	85.16%
FTAudit-Vuln-Llama3-8b	13	30	64	2	5	0	17	21	6	158	86.81% (↑ 21.98%)
FTAudit-Vuln-Gemma-7b	13	30	54	1	5	1	15	21	4	144	79.12% (↑ 36.81%)
FTAudit-Vuln-Codegemma-7b	13	30	56	1	4	1	17	22	6	148	81.32% (↑ 48.9%)
FTAudit-Vuln-Mistral-7b	14	31	60	1	6	3	18	21	6	160	87.91% (↑ 25.27%)

Note: ↑ indicates improvements compared to their base model counterparts. V1 - bad randomness, V2 - reentrancy, V3 - unchecked_low_level_calls, V4 - Other, V5 - denial-of-service, V6-front-running, V7-access control, V8 - arithmetic, V9 - time manipulation.

Moreover, much of the content in this dataset consists of code snippets rather than full smart contracts. As a result, we did not include these two datasets in our paper.

5.3 Evaluation Settings

5.3.1 Evaluation Criteria. In our study, the process of detecting vulnerabilities in smart contracts can be seen as a binary classification problem. The primary objective of our assessment tool is to accurately determine the presence or absence of specific vulnerabilities within a given smart contract. This binary classification approach simplifies our evaluation methodology and effectively measures the tool's precision in identifying vulnerabilities. The outcomes of this classification are delineated into four distinct categories:

- **True Positive (TP)**: The tool correctly identifies a vulnerability in a contract where one exists.
- **False Positive (FP)**: The tool incorrectly flags a vulnerability in a contract where none exists.
- **False Negative (FN)**: The tool fails to identify an existing vulnerability in a contract.
- **True Negative (TN)**: The tool correctly determines that a contract does not contain a vulnerability when it does not.

Thus, we introduce two metrics: a) recall (sensitivity) = TP/(TP+FN) and b) accuracy = (TP+TN)/(TP+TN+FP+FN). Recall measures the tool's ability to detect vulnerabilities when they are present. A high recall indicates that the model is good at finding vulnerabilities. Accuracy measures the overall correctness of the model's predictions, including both vulnerability detection and absence of vulnerabilities.

Notably, in our evaluation, the result is considered a TP if the "rationale" correctly identify the exact position of the vulnerability; otherwise, it is counted as a FP. We focus on the accuracy of vulnerability detection and positioning, rather than the correctness of specific vulnerability labels.

5.3.2 Experiment Setup. For model validation, we maintained consistency with our training environment by using the same NVIDIA A100 GPUs with 48GB of VRAM. To compare the performance of the base models and fine-tuned models, we expanded our evaluation to include three high performance SOTA commercial models (GPT-4o, Claude-3-Opus, and Gemini-1.5-pro). These commercial models were integrated into our evaluation pipeline through their respective public APIs. This integration allowed for a seamless comparison between our fine-tuned models and these leading commercial solutions, providing a comprehensive benchmark for our study.

5.4 Experiment Results

5.4.1 RQ1: Comparison on Detectable Dataset. To address RQ1, we present a comprehensive evaluation of base models , fine-tuned models (FTAudit-Vuln*), commercial models using the Detectable dataset. For a fair comparison, we also include the results of Slither-0.10.0 [23] and Mythril-0.24.7 [15], well-known traditional vulnerability detecting tools for Solidty smart contracts. In this evaluation, we only count the TPs and recall (sensitivity to vulnerabilities), ignoring other metrics. We focus solely on whether the labeled bugs in Detectable dataset are identified, disregarding the unlabeled bugs. The results are presented in Table 2.

The results demonstrate that all fine-tuned models showed significant improvements over their counterparts. The most substantial improvement was observed in FTAudit-Vuln-Codegemma-7b, with a 48.9% increase in accuracy compared to Codegemma-7b-it. FTAudit-Vuln-Mistral-7b achieved the highest overall accuracy at 87.91%, a 25.27% improvement over Mistral-7b-it-v0.3.

The commercial models performed well, with accuracies ranging from 80.56% to 85.16%. Gemini-1.5-pro had the highest accuracy among commercial models at 85.16%. Our best fine-tuned model (FTAudit-Vuln-Mistral-7b) slightly outperformed the best commercial model (Gemini-1.5-pro) in overall accuracy. Although both traditional tools performed well for several bug types, they were limited in their ability to detect all kinds of vulnerabilities. This resulted in overall poor performance for Slither at 66.48% and Mythril at 65.38%.

For each vulnerability type detected, we find that most models performed well in detecting reentrancy (V2) and arithmetic (V8). However, front-running (V6) and "Other" (V4) vulnerabilities were generally the most challenging to detect across all models. Fine-tuned models showed notable improvements in detecting unchecked low-level calls (V3), denial-of-service (V5), and access control (V7) vulnerabilities.

Our evaluation also revealed that the fine-tuning on both Codegemma-7b and Gemma-7b achieved nearly identical performance after fine-tuning. This suggests that the underlying architecture of both models is quite similar, leading to comparable performance. Besides, it reflect the our fine-tuning dataset might be comprehensive and of high quality, providing enough information to bring both models to a similar performance level, despite their initial differences.

> **Answer to RQ1:** Our fine-tuned models, especially FTAudit-Vuln-Mistral-7b, achieved accuracy levels comparable to or even surpassing SOTA commercial models and leading traditional detection tools. This demonstrates the effectiveness of our fine-tuning approach and underscores the potential of smaller, specialized models in improving smart contract security.

5.4.2 Comparison of Different Fine-tuning Datasets. To answer RQ2, we conducted a comparative analysis of base models, FTAudit-Vuln-* models, and FTAudit-* models using the Detectable dataset. This analysis focused on the models' ability to correctly identify vulnerabilities in vulnerable smart contracts. Table 3 presents the detailed performance metrics for each model variant.

A key observation from our analysis is the trade-off between sensitivity to vulnerabilities and accuracy in detection. These fine-tuned models, trained exclusively on vulnerable contracts, exhibited the highest TP rates but also the highest FP rates. This indicates that while they are highly sensitive to vulnerabilities, this comes at the cost of increased false positives. By including secure smart contract examples in their training data, these models generally showed a more balanced performance with lower FP rates compared to their FTAudit-Vuln-* counterparts. This suggests that exposure to secure contract examples during training helps models better distinguish between different types of vulnerabilities, potentially reducing false positives.

Table 3. Comparison of Different Fine-tuned Models

Model	TP	FP	FN	Recall	Accuracy
Llama3-8b-it	118	4	64	64.84%	63.44%
Gemma-7b-it	77	7	105	42.31%	40.74%
Codegemma-7b-it	59	3	123	32.42%	31.89%
Mistral-7b-it-v0.3	114	11	68	62.64%	59.07%
FTAudit-Vuln-Llama3-8b	158	26	24	86.81%	75.24%
FTAudit-Vuln-Gemma-7b	144	15	38	79.12%	73.1%
FTAudit-Vuln-Codegemma-7b	148	14	34	81.32%	75.51%
FTAudit-Vuln-Mistral-7b	160	62	22	**87.91%**	65.57%
FTAudit-Llama3-8b	156	17	26	85.71%	**78.39%**
FTAudit-Gemma-7b	144	10	38	79.12%	75%
FTAudit-Codegemma-7b	148	11	34	81.32%	76.68%
FTAudit-Mistral-7b	152	26	30	83.52%	73.08%

Note: In Detectable dataset, there are no TNs, as all contracts are vulnerable.

Among the evaluated models, FTAudit-Llama3-8b model achieved the highest accuracy at 78.39%, demonstrating the best trade-off between True Positives and False Positives. It outperformed other models in terms of correctly identifying vulnerabilities while maintaining a relatively lower FP rate. FTAudit-Vuln-Mistral-7b model achieved the highest recall at 87.91%, indicating the best detection ability among all models. However, it also had the lowest accuracy among the 8 fine-tuned models due to high FPs, illustrating the trade-off between high sensitivity and overall accuracy.

> **Answer to RQ2:** These results highlight the importance of balanced training data in developing models for vulnerability detection in smart contracts. While models trained exclusively on vulnerable contracts show high sensitivity, they may generate vulnerabilities in practice.

5.4.3 Performance Comparison on Contest Dataset. To address RQ3, we present a comprehensive evaluation of base models, fine-tuned models (FTAudit-*), and commercial models using the Contest dataset. The evaluation focuses on assessing the models' effectiveness in handling two categories of vulnerabilities: detectable and undetectable types. We focus on two main metrics: TPs and recall.

Table 4. Comparison of Different Models on Contest Dataset

Model	Detectable		Undetectable	
	TP	Recall	TP	Recall
GPT-4o	53	42.74%	34	13.71%
Claude-3-Opus	50	40.32%	32	12.9%
Gemini-1.5-pro	36	29.03%	39	**15.73%**
Llama3-8b-it	24	18.75%	15	6.05%
Gemma-7b-it	11	8.59%	16	6.45%
Mistral-7b-it-v0.3	21	8.59%	11	4.03%
FTAudit-Llama3-8b	80	**62.50%** (↑ 43.75%)	28	11.29% (↑ 5.24%)
FTAudit-Gemma-7b	54	42.19% (↑ 33.59%)	28	11.29% (↑ 4.84%)
FTAudit-Mistral-7b	45	35.16% (↑ 26.56%)	39	**15.73%** (↑ 11.69%)

Note: 128 detectable bugs and 248 undetectable bugs in Contest Dataset.

The results in Table 4 indicate that the fine-tuned models showed significant improvement in TPs for both detectable and undetectable vulnerabilities compared to their base model counterparts. FTAudit-Llama3-8b achieved the highest performance for detectable bugs with 80 TPs and an accuracy of 62.50%. FTAudit-Mistral-7b and Gemini-1.5-pro demonstrated the best performance on undetectable vulnerabilities, both scoring 39 TPs with an accuracy of 15.73%.

We also find that all models, both fine-tuned and commercial, performed significantly better on detectable vulnerabilities compared to undetectable ones. This disparity highlights the challenge in identifying complex, logic-based vulnerabilities. Notably, FTAudit-Llama3-8b outperformed all commercial models in detecting detectable bugs. For undetectable bugs, FTAudit-Mistral-7b matched the performance of the best commercial model (Gemini-1.5-pro).

Previous studies, such as those by David et al. [11] and GPTScan [32], have highlighted the potential of GPT-based tools for vulnerability detection. However, our research significantly extends their scope and enhances their results. For example, David et al. [11] reported a recall rate of 39.73%, detecting 58 out of 146 vulnerabilities using a combination of Claude and GPT. GPTScan analyzed 232 vulnerabilities, accurately identifying 40 true positives. In contrast, our study not only examines a broader range of vulnerability types but also achieves improved detection performance. Specifically, we analyzed 376 vulnerabilities and successfully detected 108 using FTAudit-Llama3-8b.

> **Answer to RQ3:** Our method significantly enhances model performance for well-defined vulnerability types, as evidenced by the improvement in TPs and accuracy for detectable vulnerabilities. However, the lower accuracy across all models for undetectable bugs highlights a significant challenge in smart contract security that requires further research and development. Additionally, we tested the dataset using the traditional tools Slither and Mythril. Neither Slither nor Mythril dwere able to detect any vulnerabilities within the Contest dataset.

5.4.4 Comparison on Real-World Contracts. To answer RQ4, we evaluate the performance of the fine-tuned models on the Real-world dataset. The results are presented in Figure 8. For a fair comparison, we also include the results of Slither-0.10.0. Since all contracts are unlabeled and Slither-0.10.0 can only detect the detectable bugs, we only count the detectable bugs identified.
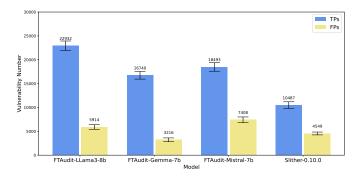

Fig. 8. Performance on Real-World Dataset

The evaluation reveals that fine-tuned models outperform Slither-0.10.0 in terms of TPs for detectable bugs, demonstrating their enhanced capability in identifying vulnerabilities in real-world contracts. Among the three models, FTAudit-Llama3-8b achieves the highest number of TPs, indicating its superior performance in identifying vulnerabilities in real-world contracts. While FTAudit-Llama3-8b and FTAudit-Gemma-7b have a higher number of TPs compared

to Slither-0.10.0, they also exhibit higher FPs. This suggests that for LLMs, higher TPs come with higher FPs, an issue that needs to be addressed in future work.

> **Answer to RQ4:** Our fine-tuned models significantly outperform Slither-0.10.0 in detecting true positives for detectable bugs. These findings highlight the effectiveness of our fine-tuned models in identifying detectable bugs in a large set of real-world contracts, providing a strong foundation for their practical application. However, the increase in false positives indicates a need for further refinement to balance sensitivity and specificity.

5.4.5 Newly Discovered Vulnerabilities. To answer RQ5, our models successfully discovered 13 vulnerabilities of 4 different types that were not identified in the audit reports from Code4rena. The vulnerabilities are listed below in Table 5. These findings have been submitted to the Code4rena community for verification.

Table 5. Summary of Potential Bugs

Issue	Location
Potential Token Lockup	earn() in yVault.sol, mint() in sYETIToken.sol
Insufficient Input Validation	setTransferRatio() in sYETIToken.sol
Unlimited Token Approval	initialize() in IdleYieldSource.sol, _supply() in YieldSourcePrizePool.sol, _maxApprove() in SingleTokenJoinV2.sol, swap() in Vault.sol, _sendForReceiver() in NFTXSimpleFeeDistributor, and lockWithPermit() in XDEFIDistribution.sol
Unprotected Function	setMinter() in Position.sol, setParams() in USDV.sol, setMin() in Vault.sol, migrate() in MapleLoan.sol

For example, one such vulnerability is a 'Potential Token Lockup'. As shown in the code snippet in YVault.sol, the 'earn' function transfers the available tokens to the controller without any checks or restrictions. If the controller contract is not properly implemented or has a vulnerability, it could potentially lock up the tokens, making them irretrievable. Another vulnerability, also a 'Potential Token Lockup', was found in sYETIToken.sol. The mint function sends YETI tokens to the sYETI contract using the 'sendToSYETI' function of the 'yetiToken' contract. If the 'yetiToken' contract has a vulnerability or is not properly implemented, it could potentially lock up the tokens, making them irretrievable.

> **Answer to RQ5:** Our fine-tuned models identified 13 new vulnerabilities that were not detected in the Code4rena audit reports. This underscores the superiority of our models over existing technologies and their effectiveness as a complement to human auditors.

This section presents a comprehensive evaluation of our fine-tuned models. We will begin by outlining the metrics and datasets used to assess the performance of the models, followed by a detailed comparison with baseline models. The results are analyzed to highlight the strengths and areas where our models excel, as well as any notable challenges or limitations encountered during the evaluation process. In the subsequent section, we will provide a thorough discussion of related works, drawing comparisons between our approach and other methodologies. Additionally, we will delve into the key findings from our experiments and address the limitations of our study, offering insights into potential future improvements.

6   RELATED WORK AND DISCUSSION

6.1   Related work

The application of LLMs in programming has been widely recognized, but their effectiveness in domain-specific languages (DSLs) such as Solidity is an emerging area of interest. Recent studies have begun to explore the potential of LLMs in the domain of smart contract security analysis.

David et al. [11] explored the application of pre-trained LLMs, including GPT-4 and Claude, for security audits of DeFi smart contracts. Their approach involved a binary classification query, which tasked the LLMs with determining the vulnerability of contracts without additional training. While GPT-4 and Claude showed a high rate of true positives, their use also led to a significant number of false positives. Furthermore, the financial burden of their evaluations was notable, costing approximately 2,000 USD to analyze 52 DeFi contracts. Chen et al. [7] conducted a comparative analysis of GPT's performance in detecting smart contract vulnerabilities against other established tools, utilizing a publicly accessible dataset. Their findings revealed that GPT's effectiveness varied across different vulnerability types, focusing on eight common ones, unlike the complex logic bugs in our study.

Similarly, GPTScan [31] tested GPT's ability to match candidate vulnerabilities using a binary response format, where GPT responds with 'Yes' or 'No' to potential matches with predefined scenarios. This study also pointed out the possibility of false positives due to GPT's inherent limitations. GPTScan analyzed 232 vulnerabilities, correctly identifying 40 true positives. However, although they reported that the advanced GPT-4 model did not yield significant improvements, our research, along with findings from other studies, clearly shows that GPT-4 significantly enhances detection capabilities. Contrary to the previous methods that relied on expensive pre-trained solutions like GPT-4 and Claude, our research leverages advanced fine-tuning of localized LLMs to achieve superior accuracy and cost-efficiency in auditing.

In addition to commercial products, open-source alternatives have been considered for smart contract analysis. Shou et al. [30] have integrated the Llama-2 model into the fuzzing process to detect vulnerabilities in smart contracts, aiming to overcome the inefficiencies of traditional fuzzing methods. While innovative, the effectiveness of this approach relies on the accurate and nuanced understanding of smart contracts by LLMs, and it faces challenges in complexity, cost, and reliance on static analysis. Moreover, recent research indicates that traditional methods such as fuzzing often fall short in addressing complex logic bugs. In contrast, our experiments have shown that directly employing SOTA models such as GPT or Llama-3 can achieve advancements in identifying these sophisticated vulnerabilities.

Sun et al. [32] explored open-source tools such as Mixtral and CodeLlama against GPT-4 for detecting smart contract vulnerabilities. They discovered that GPT-4, leveraging its advanced Assistants' functionalities to effectively utilize enhanced knowledge and structured prompts, significantly outperformed Mixtral and CodeLlama. However, the assessment relied solely on demonstrations from the Replicate website, which may not capture the full capabilities of these LLMs. Furthermore, the direct comparison between the Mixtral-8x7b-instruct and CodeLlama-13b-instruct models could be seen as skewed due to the vast differences in their parameter counts. Our study overcomes these shortcomings by conducting evaluations on models run within our local infrastructure, providing complete control over the testing environment. Additionally, we ensure a fair comparison by selecting base models with similar parameter counts, establishing a more balanced framework for assessment.

Ma et al. [24] introduced a two-stage fine-tuning approach that separates the tasks of detection and reasoning within smart contract analysis. They fine-tuned CodeLlama model and Mixtral model but did not provide specific reasons for selecting these models. Moreover, traditional models such as CodeBERT and CodeT5, which lack prior knowledge

of Solidity contracts, may find a dataset of 2,268 contract entries insufficient for effective training, especially without filtering for errors or duplicates. In contrast to their methodology, we have established clear criteria for model selection and carefully designed our fine-tuning dataset, ensuring a more robust and informed approach. Our experimental results have confirmed these advantages.

In this subsection, we review relevant studies in the field, highlighting the key distinctions and advancements of our approach. In subsequent sections, we will discuss the main findings of our research and identify potential areas for improvement.

6.2   Summary of Findings

Based on the above evaluation results, we have derived the following key findings:

- **Fine-Tuning Effectiveness**: Fine-tuning significantly enhances the accuracy of models in detecting vulnerabilities. With proper fine-tuning, small models achieve or even surpass large commercial models. This underscores the potential of fine-tuned model as a valuable tool for improving smart contract security.
- **Detectable vs. Undetectable Vulnerabilities**: All models, both fine-tuned and commercial, performed significantly better on detectable vulnerabilities compared to undetectable ones. This disparity highlights the ongoing challenge in identifying complex, logic-based vulnerabilities and the need for further research in this area.
- **Comparison with Existing Tools**: Fine-tuned models outperform traditional vulnerability detection tools like Slither-0.10.0 in identifying true positives for detectable bugs. This demonstrates the enhanced capability of our models in real-world applications, though higher false positive rates indicate a trade-off that needs to be managed.
- **Resource Efficiency**: Our fine-tuning approach, particularly using QLoRA, allows effective model training and deployment on resource-constrained environments, such as a single NVIDIA Tesla T4 GPU with 16GB memory. This showcases the feasibility of deploying advanced LLMs without requiring extensive computational resources.
- **Extensibility**: While our framework is designed with smart contract auditing as the primary application domain, it is inherently flexible and can be extended to other domains requiring LLM solutions.

6.3   Threats of Validity

Our proposed system has the following potential limitations:

- **High False Positives**: While fine-tuned models showed significant improvements in detecting vulnerabilities, they also exhibited higher false positive rates. The high rate of false positives reflects the LLM's propensity to generate a spectrum of possibilities rather than definitive answers. This characteristic, intrinsic to the probabilistic nature of LLMs, means that they offer a range of potential answers or solutions, each accompanied by a probability. This indicates we need to further improve the quality of training. Besides, fine-tuned model's ability to uncover a broad array of potential vulnerabilities provides a critical supplement to human expertise, highlighting issues that may otherwise go unnoticed in a real-world auditing context.
- **Undetectable Vulnerabilities**: Despite improvements, all models, both fine-tuned and commercial, struggled with detecting undetectable vulnerabilities, particularly those requiring deep comprehension and complex logic analysis. This highlights a gap in current capabilities and the need for advanced techniques to handle such vulnerabilities.
- **Computational Resources**: The higher the input size, the greater the computational demands on the models. During inference, larger inputs require more GPU memory, which can exceed the capabilities of the T4 GPU for very large contracts. Specifically, when a smart contract's size surpasses 4k tokens, the model requires more than 16GB of

memory, which is beyond the T4's capacity. For even larger contracts exceeding 8k tokens, the memory requirement jumps to over 24GB.

- **Other Parameter Models** In our pursuit of demonstrating that smaller models can match or even outperform larger models, we did not analyze the performance of larger parameter models within the same model family. Larger parameter models may outperform our fine-tuned smaller models, which is an aspect that requires further investigation.

7 CONCLUSION

In this paper, we introduced FTSmartAudit, a framework designed to specialize LLMs for the task of smart contract auditing. Our framework integrates four stages of creating specialized models, including data preparation, training process, evaluation, and continuous learning. We showcased the potential and effectiveness of fine-tuning LLMs to achieve high performance in detecting vulnerabilities in smart contracts. Fine-tuned models, especially FTAudit-Llama3-8b, exhibit excellent performance in detecting specific types of vulnerabilities, particularly those that are grammar-related or boundary-related. Additionally, we demonstrated that LLMs can be trained to detect complex logic vulnerabilities traditionally identified by human auditors. The quality of the fine-tuning dataset is crucial, as small models, trained on high-quality datasets, can achieve and even surpass the performance of large commercial models.

In conclusion, FTSmartAudit demonstrates the potential of fine-tuned LLMs to significantly enhance the security and reliability of smart contracts, providing a strong foundation for their practical application in real-world scenarios. Our future work will focus on addressing these limitations by enhancing the models' ability to handle complex vulnerabilities and lowering false positives. Additionally, continuous updates and retraining will be necessary to adapt to the dynamic and evolving nature of smart contract development. Moreover, we find that LLMs can not only replace the work of human auditors in identifying security issues but also operate other tools to perform complex attacks on blockchain or smart contracts. Furthermore, LLMs have the potential to fix vulnerable contracts and upgrade them to secure versions.

REFERENCES
[1] 2022. SB Curated: A Curated Dataset of Vulnerable Solidity Smart Contracts. https://github.com/smartbugs/ smartbugs-curated.
[2] 2022. Web3Bugs. https://github.com/ZhangZhuoSJTU/ Web3Bugs.
[3] Zhangir Azerbayev, Ansong Ni, Hailey Schoelkopf, and Dragomir Radev. 2022. Explicit Knowledge Transfer for Weakly-Supervised Code Generation. *arXiv preprint arXiv:2211.16740* (2022).
[4] Shraddha Barke, Michael B James, and Nadia Polikarpova. 2023. Grounded copilot: How programmers interact with code-generating models. *Proceedings of the ACM on Programming Languages* 7, OOPSLA1 (2023), 85–111.
[5] Lucas Beyer, Xiaohua Zhai, Amélie Royer, and Larisa Markeeva. 2022. Knowledge distillation: A good teacher is patient and consistent. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*. 10925–10934.
[6] Dan Biderman, Jose Gonzalez Ortiz, Jacob Portes, et al. 2024. LoRA Learns Less and Forgets Less. *arXiv preprint arXiv:2405.09673* (2024).
[7] Chong Chen, Jianzhong Su, Jiachi Chen, et al. 2023. When ChatGPT Meets Smart Contract Vulnerability Detection: How Far Are We? *arXiv preprint arXiv:2309.05520* (2023).
[8] Huashan Chen, Marcus Pendleton, Laurent Njilla, and Shouhuai Xu. 2020. A survey on ethereum systems security: Vulnerabilities, attacks, and defenses. *ACM Computing Surveys (CSUR)* 53, 3 (2020), 1–43.
[9] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374* (2021).
[10] Yizheng Chen, Zhoujie Ding, Lamya Alowain, et al. 2023. Diversevul: A new vulnerable source code dataset for deep learning based vulnerability detection. In *Proceedings of the 26th International Symposium on Research in Attacks, Intrusions and Defenses*. 654–668.
[11] Isaac David, Liyi Zhou, Kaihua Qin, et al. 2023. Do you still need a manual smart contract audit? *arXiv preprint arXiv:2306.12338* (2023).
[12] Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, and Luke Zettlemoyer. 2024. Qlora: Efficient finetuning of quantized llms. *Advances in Neural Information Processing Systems* 36 (2024).
[13] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805* (2018).

[14] Yangruibo Ding, Marcus J Min, Gail Kaiser, and Baishakhi Ray. 2024. Cycle: Learning to self-refine the code generation. *Proceedings of the ACM on Programming Languages* 8, OOPSLA1 (2024), 392–418.

[15] Josselin Feist, Gustavo Grieco, and Alex Groce. 2019. Slither: a static analysis framework for smart contracts. In *2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*. IEEE, 8–15.

[16] Jianping Gou, Baosheng Yu, Stephen J Maybank, and Dacheng Tao. 2021. Knowledge distillation: A survey. *International Journal of Computer Vision* 129, 6 (2021), 1789–1819.

[17] Jingxuan He and Martin Vechev. 2023. Large language models for code: Security hardening and adversarial testing. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*. 1865–1879.

[18] Cheng-Yu Hsieh, Chun-Liang Li, Chih-Kuan Yeh, et al. 2023. Distilling Step-by-Step! Outperforming Larger Language Models with Less Training Data and Smaller Model Sizes. In *The 61st Annual Meeting Of The Association For Computational Linguistics*.

[19] Guohao Li, Hasan Hammoud, Hani Itani, et al. 2024. Camel: Communicative agents for" mind" exploration of large language model society. *Advances in Neural Information Processing Systems* 36 (2024).

[20] Haonan Li, Yu Hao, Yizhuo Zhai, and Zhiyun Qian. 2024. Enhancing Static Analysis for Practical Bug Detection: An LLM-Integrated Approach. *Proceedings of the ACM on Programming Languages* 8, OOPSLA1 (2024), 474–499.

[21] Raymond Li, Loubna Ben Allal, Yangtian Zi, et al. 2023. Starcoder: may the source be with you! *arXiv preprint arXiv:2305.06161* (2023).

[22] Bingchang Liu, Chaoyu Chen, Cong Liao, et al. 2023. Mftcoder: Boosting code llms with multitask fine-tuning. *arXiv preprint arXiv:2311.02303* (2023).

[23] Pengxiang Ma, Ningyu He, Yuhua Huang, et al. 2023. Abusing the Ethereum Smart Contract Verification Services for Fun and Profit. *arXiv preprint arXiv:2307.00549* (2023).

[24] Wei Ma, Daoyuan Wu, Yuqiang Sun, et al. 2024. Combining Fine-Tuning and LLM-based Agents for Intuitive Smart Contract Auditing with Justifications. *arXiv preprint arXiv:2403.16073* (2024).

[25] Subhabrata Mukherjee, Arindam Mitra, Ganesh Jawahar, et al. 2023. Orca: Progressive learning from complex explanation traces of gpt-4. *arXiv preprint arXiv:2306.02707* (2023).

[26] Colin Raffel, Noam Shazeer, Adam Roberts, et al. 2020. Exploring the limits of transfer learning with a unified text-to-text transformer. *Journal of Machine Learning Research* 21, 140 (2020), 1–67.

[27] Nils Reimers and Iryna Gurevych. 2019. Sentence-bert: Sentence embeddings using siamese bert-networks. *arXiv preprint arXiv:1908.10084* (2019).

[28] Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, et al. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950* (2023).

[29] Victor Sanh, Albert Webson, Colin Raffel, et al. 2021. Multitask prompted training enables zero-shot task generalization. *arXiv preprint arXiv:2110.08207* (2021).

[30] Chaofan Shou, Jing Liu, Doudou Lu, and Koushik Sen. 2024. LLM4Fuzz: Guided Fuzzing of Smart Contracts with Large Language Models. *arXiv preprint arXiv:2401.11108* (2024).

[31] Yuqiang Sun, Daoyuan Wu, Yue Xue, et al. 2024. Gptscan: Detecting logic vulnerabilities in smart contracts by combining gpt with program analysis. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (ICSE)*. 1–13.

[32] Yuqiang Sun, Daoyuan Wu, Yue Xue, et al. 2024. LLM4Vuln: A Unified Evaluation Framework for Decoupling and Enhancing LLMs' Vulnerability Reasoning. *arXiv preprint arXiv:2401.16185* (2024).

[33] CodeGemma Team. 2024. Codegemma: Open code models based on gemma. *arXiv preprint arXiv:2406.11409* (2024).

[34] Palina Tolmach, Yi Li, Shang-Wei Lin, et al. 2021. A survey of smart contract formal specification and verification. *ACM Computing Surveys (CSUR)* 54, 7 (2021), 1–38.

[35] Shuohang Wang, Yang Liu, Yichong Xu, Chenguang Zhu, and Michael Zeng. 2021. Want to reduce labeling cost? GPT-3 can help. *arXiv preprint arXiv:2108.13487* (2021).

[36] Jason Wei, Maarten Bosma, Vincent Y Zhao, et al. 2021. Finetuned language models are zero-shot learners. *arXiv preprint arXiv:2109.01652* (2021).

[37] Zhiyuan Wei, Jing Sun, Zijian Zhang, Xianhao Zhang, Xiaoxuan Yang, and Liehuang Zhu. 2023. Survey on quality assurance of smart contracts. *ACM Computing Surveys (CSUR)* (2023).

[38] Can Xu, Qingfeng Sun, Kai Zheng, et al. 2023. Wizardlm: Empowering large language models to follow complex instructions. *arXiv preprint arXiv:2304.12244* (2023).

[39] Chenyuan Zhang, Hao Liu, Jiutian Zeng, et al. 2024. Prompt-enhanced software vulnerability detection using chatgpt. In *Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings*. 276–277.

[40] Jialu Zhang, José Pablo Cambronero, Sumit Gulwani, Vu Le, Ruzica Piskac, Gustavo Soares, and Gust Verbruggen. 2024. Pydex: Repairing bugs in introductory python assignments using llms. *Proceedings of the ACM on Programming Languages* 8, OOPSLA1 (2024), 1100–1124.

[41] Zhuo Zhang, Brian Zhang, Wen Xu, and Zhiqiang Lin. 2023. Demystifying exploitable bugs in smart contracts. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 615–627.

[42] Andrew Zhao, Daniel Huang, Quentin Xu, et al. 2024. Expel: Llm agents are experiential learners. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 38. 19632–19642.

[43] Wayne Xin Zhao, Kun Zhou, Junyi Li, et al. 2023. A survey of large language models. *arXiv preprint arXiv:2303.18223* (2023).

[44] Lianmin Zheng, Zhuohan Li, Hao Zhang, et al. 2022. Alpa: Automating inter-and {Intra-Operator} parallelism for distributed deep learning. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. 559–578.

[45] Chunting Zhou, Pengfei Liu, Puxin Xu, et al. 2024. Lima: Less is more for alignment. *Advances in Neural Information Processing Systems* 36 (2024).
[46] Liyi Zhou, Xihan Xiong, Jens Ernstberger, et al. 2023. Sok: Decentralized finance (defi) attacks. In *2023 IEEE Symposium on Security and Privacy (S&P)*. IEEE, 2444–2461.
[47] Bojia Zi, Xianbiao Qi, Lingzhi Wang, et al. 2023. Delta-lora: Fine-tuning high-rank parameters with the delta of low-rank matrices. *arXiv preprint arXiv:2309.02411* (2023).

APPENDICES

A   EXPERIMENTAL PROCESS FOR MODEL SELECTION AND FINE-TUNING

A.1   Initial Model Evaluation

The first pattern in the initial filtering involves selecting basic models from a variety of open-source LLMs. Our target models include: CodeBERT-base, GraphCodeBERT-base, CodeT5-base, UnixCoder-base, StarCode2-7b, Llama2-7b, Codellama-7b, Yi-6b, Solar-10.7b, Phi-2, Llama3-8b, Gemma-7b, CodeGemma-7b and Mistral-7b. Our evaluation dataset, the initial fine-tuning dataset $\mathcal{D}(0)$, contains 500 vulnerable contracts ($N = 500$).

To calculate the loss $\mathcal{L}_{label}$, we use the following equation:

$$\mathcal{L}_{label} = -\frac{1}{N} \sum_{i=1}^{N} \log p(y_i|f(x_i))$$

where N represents the total number of input data points, and $p(y_i|f(x_i))$ is the probability that the model assigns to the correct label y_i given the input x_i.

Let p_i denote the probability that the model assigns to the correct label for each input. Without specific probabilities for each correct prediction, we assume a general average probability p for correct predictions and $1 - p$ for incorrect ones.

Given that N_{co} represents correct number and N_{in} is incorrect number, the loss function $\mathcal{L}_{label}$ can be divided into the contributions from correct and incorrect predictions.

$$\mathcal{L}_{label} = -\frac{1}{N} \left(\sum_{i=1}^{N_{co}} \log p_i + \sum_{i=N_{co}+1}^{N} \log(1 - p_i) \right)$$

However, without specific probabilities p_i for each data point, we assume an average probability:

- For correct predictions: $p(y_i|f(x_i)) = p$
- For incorrect predictions: $p(y_i|f(x_i)) = 1 - p$

Substituting these into the equation, we get:

$$\mathcal{L}_{label} = -\frac{1}{N} \left(N_{co} \cdot \log(p) + N_{in} \cdot \log(1 - p) \right)$$

In a typical machine learning scenario, particularly with models that are not yet well-trained, the probability p assigned to the correct labels might not be very high. For classification problems a reasonable assumption might be that the model's average confidence in the correct predictions is around 70%. This translates to $p = 0.7$.

$$\mathcal{L}_{label} = -\frac{1}{N} \left(N_{co} \cdot \log(0.7) + N_{in} \cdot \log(0.3) \right)$$

Thus, when $N_{co} = 0$, $\mathcal{L}_{label} = 1.20$, and when $N_{co} = 500$, $\mathcal{L}_{label} = 0.35$. By setting the initial loss function $\mathcal{L}_b = 1.12$ (when $N_{co} = 50$), we establish a baseline performance measure. This indicates that while the model has some ability to correctly predict vulnerabilities in Solidity smart contracts, there is significant room for improvement.

Table 6. Initial Model Performance

Model	$\mathcal{L}_{label}$	Model	$\mathcal{L}_{label}$
CodeBERT-base	1.20	GraphCodeBERT-base	1.20
CodeT5-base	1.20	UnixCoder-base	1.20
Yi-6b	1.20	Phi-2	1.20
Solar-10.7b	1.17	StarCode2-7b	1.14
Llama2-7b	1.01	Codellama-7b	0.97
qwen2-7b	1.03	Starling-LM-7b	1.01
Gemma-7b	1.04	CodeGemma-7b	1.01
Mistral-7b	0.97	llama3-8b	0.92

The initial model performance is list in Table 6. We then select the models: llama2-7b, codellama-7b,qwen2-7b, Starling-LM-7b, llama3-8b, Gemma-7b, CodeGemma-7b and Mistral-7b, which is lower than the baseline of $\mathcal{L}_b = 1.06$.

A.2 Iterative Enhancement Process

In this next step, we focus on improving both model performance and data quality through an iterative process. This involves integrating label prediction loss $\mathcal{L}_{label}$ with rationale prediction loss $\mathcal{L}_{rationale}$, using a dynamic weighting scheme controlled by the hyperparameter λ. By tuning λ, we can adjust the model's focus on correctly predicting rationales versus labels.

Given that rationale predictions are essential for understanding and explaining the model's decisions, but label accuracy remains the primary objective, a reasonable starting value might be: $\lambda = 0.7$. This indicates that generating correct rationales is given greater importance.

Each model was evaluated by computing $\mathcal{L}_{label}$ and $\mathcal{L}_{rationale}$, followed by the overall loss $\mathcal{L}$:

$$\mathcal{L} = \mathcal{L}_{label} + 0.7 * \mathcal{L}_{rationale}$$

To calculate the rationale prediction loss $\mathcal{L}_{rationale}$ using Equation (2), we need the predicted probabilities $g(x_i)$ for each input i, which is a value between 0 and 1. After the initial model evaluation, we assumue all the models is confident, and we apply a uniform assumption:

- For correct outputs: $g(x_i) \approx 0.8$
- For incorrect outputs: $1 - g(x_i) \approx 0.2$

Given that N_{co} is correct number and N_{in} is incorrect number. Then, we get

$$\mathcal{L}_{rationale} = -\frac{1}{N}(N_{co} * log(0.8) + N_{in} * log(0.2))$$

The experiment followed an iterative process to evaluate and refine the models:

Threshold Evaluation: Models were categorized based on their $\mathcal{L}$ values relative to the thresholds $\mathcal{L}_l$ and $\mathcal{L}_h$:

- $\mathcal{L} \leq \mathcal{L}_l$: Model is well-optimized.
- $\mathcal{L}_l < \mathcal{L} \leq \mathcal{L}_h$: Model is acceptable but may require further refinement.
- $\mathcal{L} > \mathcal{L}_h$: Model is unsuitable.

We set $\mathcal{L}_h = 1.74$ (with $\mathcal{L}_{label} = 0.93$ and $\mathcal{L}_{rationale} = 1.1$) and $\mathcal{L}_l = 0.84$ (with $\mathcal{L}_{label} = 0.51$ and $\mathcal{L}_{rationale} = 0.47$).

Model Performance: Table 7 summarizes the performance of each model at the fist round. Besides, the learning rate of SFT is fixed as 5e-4.

Model Selection: The selected models are Gemma-7b, CodeGemma-7b, Mistral-7b, and llama3-8b.

Table 7. The First Round Model Performance

Model	$\mathcal{L}_{label}$	$\mathcal{L}_{rationale}$	$\mathcal{L}$	Evaluation
Codellama-7b	1.02	1.05	1.745	Unsuitable
llama2-7b	1.01	1.08	1.766	Unsuitable
qwen2-7b	1.04	1.23	1.901	Unsuitable
Starling-LM-7b	1.02	1.31	1.937	Unsuitable
Gemma-7b	0.88	1.1	1.65	Acceptable
CodeGemma-7b	0.91	1.1	1.68	Acceptable
Mistral-7b	0.8	0.92	1.444	Acceptable
Llama3-8b	0.78	0.85	1.375	Acceptable

Dataset Upgrade and LoRA Weights: After the first round, we can find that although four models are well-optimized, none can meet the $\mathcal{L} \le \mathcal{L}_l$ criterion. We then adjust the fine-tuning data $\mathcal{D}$ and the LoRA weights until all the models meet the $\mathcal{L} \le \mathcal{L}_l$ criterion.

Finally, we get a vulnerable dataset that contains 973 contracts. Table 8 summarizes the performance of each model in the final round.

Table 8. Final Model Performance

Model	$\mathcal{L}_{label}$	$\mathcal{L}_{rationale}$	$\mathcal{L}$	Evaluation
Gemma-7b	0.46	0.39	0.733	Well-Optimized
CodeGemma-7b	0.44	0.36	0.692	Well-Optimized
Mistral-7b	0.42	0.33	0.651	Well-Optimized
Llama3-8b	0.4	0.30	0.61	Well-Optimized