Efficient Adaptation of Large Language Models for Smart Contract Vulnerability Detection

Fadul Sikder
The University of Texas at Arlington
Arlington, USA
fxs5359@mavs.uta.edu

Yu Lei
The University of Texas at Arlington
Arlington, USA
ylei@cse.uta.edu

Yuede Ji
The University of Texas at Arlington
Arlington, USA
yuede.ji@uta.edu

Abstract

Smart contracts underpin decentralized applications but face significant security risks from vulnerabilities, while traditional analysis methods have limitations. Large Language Models (LLMs) offer promise for vulnerability detection, yet adapting these powerful models efficiently, particularly generative ones, remains challenging. This paper investigates two key strategies for the efficient adaptation of LLMs for Solidity smart contract vulnerability detection: (1) replacing token-level generation with a dedicated classification head during fine-tuning, and (2) selectively freezing lower transformer layers using Low-Rank Adaptation (LoRA). Our empirical evaluation demonstrates that the classification head approach enables models like Llama 3.2 3B to achieve high accuracy (77.5%), rivaling the performance of significantly larger models such as the fine-tuned GPT-3.5. Furthermore, we show that selectively freezing bottom layers reduces training time and memory usage by approximately 10-20% with minimal impact on accuracy. Notably, larger models (3B vs. 1B parameters) exhibit greater resilience to layer freezing, maintaining high accuracy even with a large proportion of layers frozen, suggesting a localization of general code understanding in lower layers versus task-specific vulnerability patterns in upper layers. These findings present practical insights for developing and deploying performant LLM-based vulnerability detection systems efficiently, particularly in resource-constrained settings.

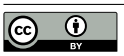
CCS Concepts

• **Security and privacy → Domain-specific security and privacy architectures**; **Distributed systems security**; • **Computing methodologies → Natural language processing**; *Supervised learning by classification.*

Keywords

Large Language Models, Smart Contracts, Vulnerability Detection, Fine-tuning, Transfer Learning, Solidity, Layer Freezing, StarCoder, Llama 3.2

ACM Reference Format:
Fadul Sikder, Yu Lei, and Yuede Ji. 2025. Efficient Adaptation of Large Language Models for Smart Contract Vulnerability Detection. In *21st International Conference on Predictive Models and Data Analytics in Software Engineering (PROMISE '25), June 26, 2025, Trondheim, Norway*. ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3727582.3728688



PROMISE '25, Trondheim, Norway

ACM ISBN 979-8-4007-1594-5/25/06
https://doi.org/10.1145/3727582.3728688

1 Introduction

Blockchain technology has revolutionized various sectors by enabling secure, transparent, and decentralized transactions [24]. A key application of this innovation is smart contracts, self-executing contracts with the terms of the agreement directly written into code. These programmatic agreements have found applications across diverse domains, including finance, supply chain management, and digital identity verification [42].

Smart contracts, particularly on platforms like Ethereum, have experienced exponential growth in recent years, with the total value locked (TVL) in decentralized finance (DeFi) smart contracts alone surpassing $50 billion as of 2023 [2]. This rapid adoption, however, has brought with it a significant rise in security vulnerabilities and exploits. The immutable nature of blockchain, while foundational to its trust model, exacerbates the impact of any flaws in smart contracts. Once deployed, a vulnerable contract cannot be easily patched, making it an attractive target for malicious actors [18].

The consequences of smart contract vulnerabilities can be severe. In 2023 alone, hacking incidents involving smart contracts resulted in approximately $1.84 billion in financial losses across 751 incidents [6]. These breaches not only cause substantial financial damage but also lead to legal disputes and undermine trust in blockchain technology. Given the high stakes and potential for significant reputational damage, ensuring the security and correctness of smart contracts has become a critical priority for developers, auditors, and blockchain platforms alike.

Traditional approaches to smart contract security analysis, broadly categorized into static and dynamic analysis, face significant limitations. Static analysis techniques, particularly symbolic execution and model checking, attempt to systematically explore program states by analyzing code without execution. While symbolic execution can theoretically provide complete coverage by exploring all possible execution paths, it encounters severe scalability challenges when dealing with smart contracts due to the state explosion problem and complex environmental dependencies [33, 34]. Dynamic analysis methods, including fuzz testing, must navigate a vast and complex state space with concrete input, making comprehensive exploration challenging. Both static and dynamic approaches often rely on expert-driven heuristics to narrow the search space, which can lead to oversights in vulnerability detection [27]. Furthermore, the rigidity of predefined rules in static analysis tools frequently results in high rates of false positives and negatives, as they struggle to adapt to the rapid emergence of new vulnerability patterns in the blockchain ecosystem [31].

Machine learning models for smart contract vulnerability detection have made significant strides, offering notable improvements over traditional rule-based methods. However, many approaches,

particularly graph-based neural networks and NLP-based models, require substantial feature engineering, such as converting source code into graph representations or extracting specific code properties, which constrains their ability to adapt and generalize to new or complex vulnerabilities. Additionally, the effectiveness of these models can vary significantly based on the quality and comprehensiveness of the engineered features, leading to inconsistent performance across different types of smart contracts and vulnerability classes [37].

Building on the challenges and constraints of feature-engineered machine learning solutions, large language models (LLMs) have emerged as a promising alternative for smart contract security analysis. Since this class of neural networks is designed to handle tasks involving long-sequence processing, LLMs can directly analyze Solidity source code with minimal preprocessing, leveraging their broad language understanding to identify subtle vulnerability patterns [15]. Broadly, two core approaches have been explored for using LLMs in smart contract vulnerability detection: prompt-based inference and fine-tuning-based adaptation.

In the prompt-based paradigm, researchers query a pretrained model (e.g., GPT-3.5, GPT-4) using carefully designed instructions to extract vulnerability-related insights without modifying its internal parameters. This method enables rapid deployment and supports various vulnerability types. Examples include ChatGPT's chain-of-thought prompts [7] and GPT-4–assisted logic checks [8], which allow the LLM to "reason" about potential flaws in a contract. However, prompt-based approaches frequently suffer from model hallucinations, high false-positive rates, and recall limitations when faced with novel or complex exploit scenarios [8, 30]. Additionally, performance is highly sensitive to prompt quality—slight variations in phrasing can lead to inconsistent results.

In contrast, fine-tuning-based approaches address these shortcomings by training or partially re-training LLMs on curated vulnerability datasets. This process aligns the model's internal representations more closely with Solidity semantics and common exploit patterns, significantly improving recall and precision [1, 41]. However, fine-tuning comes at the cost of substantial computational overhead and requires comprehensive, labeled vulnerability datasets. Despite these challenges, fine-tuned models demonstrate enhanced robustness, emphasizing the importance of updating model weights and adapting LLMs for domain-specific tasks such as Solidity smart contract vulnerability classification.

A prominent challenge in fine-tuning large language models (LLMs) for classification tasks is reconciling the generative nature of many popular architectures (e.g., decoder-only GPT variants) with the requirement to select outputs from a fixed set of class labels. In most fine-tuning workflows, the model is trained on a supervised dataset to generate the next token corresponding to the correct label (or label words) for each input example [20]. This token-level classification approach leverages the model's existing decoder to output a label token from its vocabulary set (e.g., "Vulnerable," "Safe," "Overflow"). While this method is direct and intuitive, it poses a significant risk: the decoder may generate irrelevant tokens that do not match the desired class labels. To mitigate this, researchers often impose constraints on token generation or apply post-processing techniques to map the output to the nearest recognized label [12]. Beyond irrelevant token generation, LLMs are inherently prone to hallucination—the generation of content that appears plausible but is factually incorrect or unfounded [3]. Even when generating class labels, the model may occasionally "hallucinate" labels that do not belong to the predetermined set, especially when faced with ambiguous inputs or edge cases outside their training distribution.

An alternative approach treats the LLM's final hidden representations as feature vectors rather than relying on its decoder to produce a label token [12]. Instead of generating class names as the next word, a custom classification head—such as a fully connected layer—is added on top of the model's final (or penultimate) Transformer layer. This method replaces token-level classification with a standard softmax layer trained on the model's output embeddings, transforming the LLM into a feature-extractor-plus-classifier pipeline. The classification head outputs discrete class probabilities for each vulnerability type, reducing the likelihood of hallucinated labels outside of recognized classes. Studies suggest that this approach improves accuracy in domain-specific tasks by leveraging the semantic abstractions learned by the LLM's encoder or decoder blocks while avoiding extraneous token generation [11, 12].

Fine-tuning an LLM for specialized tasks like smart contract security introduces computational and architectural challenges. Fully retraining every layer is expensive. To mitigate these issues, parameter-efficient fine-tuning methods such as LoRA [14] and QLoRA [9] have been developed. These techniques freeze most model weights and update only small adapter modules or low-rank parameter offsets, significantly reducing computational overhead. A related strategy, layer-wise fine-tuning, explores whether freezing maintains (or even enhances) domain performance [19]. From a transfer learning perspective—which underpins modern LLM fine-tuning—it is crucial to determine which portions of the model require adaptation for a new domain. Lower layers typically capture general syntactic and lexical patterns, while upper layers encode more domain- and task-specific features [11, 39]. This raises a key question: What happens if we freeze the lower layers entirely and fine-tune only the top few layers (along with a classification head)? If effective, this approach could indicate that LLMs can be fine-tuned in a more specialized way for domain-specific tasks like security vulnerability detection in Solidity smart contracts. Additionally, it could reduce training costs without sacrificing accuracy, making it an efficient strategy for domain-specific adaptation.

In this paper, we address these challenges by investigating the effectiveness of different fine-tuning configurations for large language models in the context of smart contract vulnerability detection. Our research is guided by two key research questions that examine the trade-offs between different classification approaches and layer-wise fine-tuning strategies. These questions aim to provide insights into reducing the computational requirements for LLM-based vulnerability detection systems without significant compromise on performance. By systematically evaluating these approaches, we contribute to the ongoing discourse on adapting large language models for specialized security tasks in the blockchain domain. Below are the two key research questions we aim to answer:

RQ1: Is fine-tuning an LLM with a dedicated classifier head effective for detecting smart contract vulnerabilities?

We investigate how adding a dedicated classification layer on top of the LLM's final hidden representations affects vulnerability detection accuracy.

RQ2: Does restricting fine-tuning to specific high-level Transformer layers (while freezing lower layers) achieve competitive performance on vulnerability classification?

Here, we investigate whether layer-wise fine-tuning (combined with a classification head) can adapt the LLM's capabilities to domain specific tasks like smart contract vulnerability detection while reducing computational overhead.

The remainder of this paper is organized as follows: Section 2 provides background on LLMs and fine-tuning relevant to smart contract security. Section 3 details our methodology, including model selection, the classification head architecture, datasets, and evaluation strategy. Section 4 presents the experimental setup, results, and discussion for both research questions. Section 5 reviews related work in LLM-based vulnerability detection, and Section 6 concludes with our findings and future research directions.

2 Background

2.1 Large Language Models

Large Language Models (LLMs), based on transformer-based neural network architectures, have emerged as a transformative force in natural language processing (NLP) and, more recently, in software development tasks such as code generation, completion, and debugging. LLMs are characterized by their large scale, often consisting of billions of parameters, which enables them to capture complex patterns and relationships within vast amounts of textual data [5]. The transformer architecture, introduced by Vaswani et al. [32], allows efficient parallel processing of input sequences using self-attention mechanisms, making it particularly effective in capturing long-range dependencies in text—a key feature for generating coherent and contextually relevant content.

LLMs typically undergo a two-phase development process: pre-training and fine-tuning. Pre-training is an self-supervised learning process where models are exposed to extensive corpora of text, including diverse sources such as books, websites, and code repositories. This exposure enables the model to develop a broad understanding of language structure, semantics, and domain-specific knowledge. The pre-training objective is usually autoregressive, where the model learns to predict the next token in a sequence based on the previous tokens [26].

For particular downstream tasks like smart contract vulnerability detection, pre-trained LLMs are further refined through fine-tuning. This process involves additional training on smaller, task-specific label datasets, enabling the model to specialize in particular applications. Fine-tuning can involve updating all model parameters or, in more resource-efficient methods, adjusting only a subset of parameters while keeping most of the pre-trained weights frozen [13].

2.2 Transfer Learning and Fine-Tuning

The adaptation of pre-trained models via transfer learning and fine-tuning is a cornerstone of modern deep learning. Foundational work, particularly in computer vision, established that lower layers of deep networks often learn general features (e.g., edges and textures) transferable across tasks, while upper layers capture more task-specific information [39]. This insight motivated early fine-tuning strategies involving freezing lower layers and retraining only the upper ones for new domains [39].

The evolution continued with methods addressing challenges like overfitting on small target datasets and catastrophic forgetting—where general knowledge is lost during specialized training [22]. Frameworks like ULMFiT [13] introduced gradual unfreezing techniques, while models like BERT [10] demonstrated the success of full end-to-end fine-tuning on task-specific data. Research also indicated that the optimal strategy (feature extraction vs. full fine-tuning) can depend on the similarity between the pre-training and target tasks [25].

With the advent of extremely large models (e.g., GPT-3 with 175B parameters [5]), the computational cost of full fine-tuning necessitated more efficient approaches, as noted in the Introduction. Parameter-Efficient Fine-Tuning (PEFT) methods, such as Low-Rank Adaptation (LoRA) [14] and its quantized variant QLoRA [9], significantly reduce resource needs by updating only small, low-rank adapters instead of all weights. Alongside PEFT, selective or layer-wise fine-tuning—updating only a subset of layers—has proven effective. For instance, studies like Lee et al. [19] have shown that fine-tuning only the top layers (e.g., the top 25% in BERT or RoBERTa) can retain a large fraction (90%) of the performance achieved by full fine-tuning, while substantially lowering training costs. Such strategies help preserve the model's core linguistic or domain knowledge encoded in lower layers while efficiently specializing the upper layers for the target task, also mitigating catastrophic forgetting [22].

This overall trend towards balancing performance with computational efficiency is highly relevant for adapting LLMs to specialized domains like smart contract vulnerability detection. Although Solidity presents unique syntax and semantics, it builds on general programming concepts potentially learned during pre-training. Therefore, fine-tuning approaches that leverage this pre-existing knowledge by selectively adapting higher-level representations to identify vulnerability patterns, such as the layer-freezing techniques investigated in this paper, are particularly promising.

3 Methodology

This section outlines our methodology for adapting Large Language Models (LLMs) to detect smart contract vulnerabilities. We explain the rationale guiding our key decisions concerning four primary components: (1) model selection criteria and process, (2) architectural modifications for classification, (3) dataset selection and preparation, and (4) evaluation strategies. This section documents the analytical process that informed our experimental design, aimed at addressing the research questions posed earlier.

3.1 Model Selection

Selecting an appropriate Large Language Model (LLM) for smart contract vulnerability detection requires careful consideration of several key factors: context window size, model parameters (computational footprint), and domain-specific training data (Solidity inclusion). These factors significantly impact the model's ability to process and analyze complex smart contracts effectively.

A large context window is crucial for smart contract analysis, as vulnerabilities often arise from interactions spanning different code sections, potentially across lengthy contracts. Analysis of typical

Table 1: Comparison of LLM characteristics relevant to smart contract analysis

Model	Parameters	Max Context Length	Solidity in Training
Llama-3-8B	8B	8,000 tokens	Unknown
Llama-3.2-1B	1B	128,000 tokens	Unknown
Llama-3.2-3B	3B	128,000 tokens	Unknown
CodeLlama-7b	7B	16,000 tokens	Unknown
Phi-3-mini-4k	3.8B	4,000 tokens	Unknown
StarCoderBase-1b	1B	8,192 tokens	Yes
StarCoderBase-3b	3B	8,192 tokens	Yes
StarCoderBase-7b	7B	8,192 tokens	Yes
GPT-3	175B	4,096 tokens	Unknown
GPT-J	6B	2,048 tokens	Unknown
GPT-2	774M	1,024 tokens	Unknown
CodeT5	220M	512 tokens	No

Solidity contract lengths reveals this need; using the StarCoder tokenizer, approximately 27% of contracts in a representative dataset subset exceeded 2,000 tokens, and about 12% surpassed 4,000 tokens (Figure 1). This makes models with limited context windows (e.g., GPT-2: 1,024, CodeT5: 512 tokens) impractical without significant preprocessing that could obscure context. Therefore, models capable of handling sequences exceeding 4,000 tokens were deemed necessary.

However, larger model parameters and context windows increase computational demands. While prior work highlights resource-intensive requirements even for moderately sized models (e.g., Storhaug et al. [29] reported needing multiple high-end GPUs and extended training time for a 6B parameter model), our primary constraint stemmed from direct experience. Attempts to fine-tune models with 7B or more parameters (such as StarCoder-7B, LLaMA 3-8B, or CodeLlama-7b) on our target single NVIDIA A100 40GB GPU consistently failed due to memory limitations when processing the extended sequence lengths typical of smart contracts. This established a practical upper bound of 3B parameters for our experimental setup.

Another crucial factor is the inclusion of domain-specific training data. Models pre-trained on datasets containing Solidity source code are likely to better understand its specific syntax and semantics, potentially improving vulnerability detection.

Another crucial factor in model selection is the composition of the training data, particularly the inclusion of Solidity source code. Models pre-trained on datasets that include Solidity code are likely to have a better understanding of smart contract-specific syntax and semantics, potentially leading to more accurate vulnerability detection. While other powerful models exist (e.g., GPT family, Claude, DeepSeek, Qwen), our selection was guided by fine-tuning requirements and accessibility. API-only models (Claude, GPT) could not be used for our layer-wise fine-tuning experiments. Others like DeepSeek or Qwen were either unavailable during our research timeframe or subject to institutional usage restrictions. Our primary focus remains on investigating generalizable fine-tuning techniques (classification head adaptation, layer-freezing) applicable to open-source models, rather than an exhaustive comparison of all LLMs.

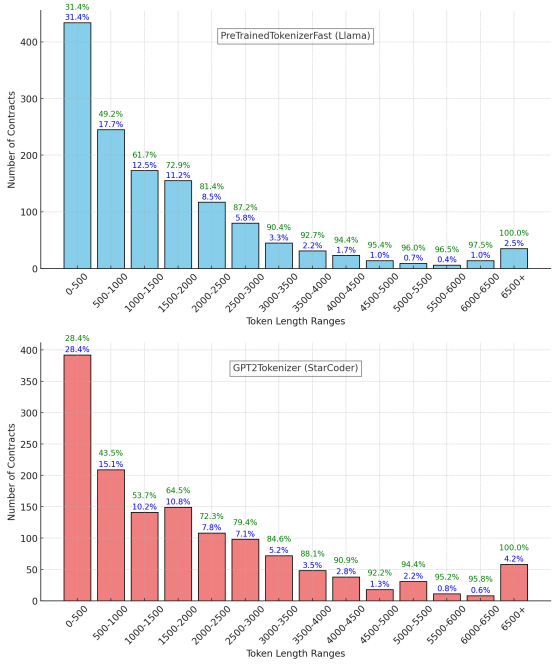


Figure 1: Distribution of token lengths in the filtered Solidity contract dataset using two different tokenizers.

Table 2: Comparative analysis of selected LLMs for smart contract vulnerability detection

Model	Context Window	Parameters / Footprint	Domain Training (Solidity)
Llama-3.2 1B	+ Very Large (128k)	+ Small (1B) + Fits 40GB GPU – Lower capacity	? Likely exposure
Llama-3.2 3B	+ Very Large (128k)	+ Medium (3B) + Fits 40GB GPU – 3x cost vs 1B	? Likely exposure
StarCoder 1B	+ Good (8k)	+ Small (1B) + Fits 40GB GPU – Lower accuracy	+ Explicitly included
StarCoder 3B	+ Good (8k)	+ Medium (3B) + Fits 40GB GPU – 3x cost vs 1B	+ Explicitly included

Based on these considerations—balancing context window size, computational feasibility (parameters ≤ 3B for our hardware), and potential Solidity pre-training, as summarized in Table 2—we selected Llama 3.2 (1B and 3B) and StarCoder (1B and 3B) for our experiments. This choice allows comparing models with very large context windows but uncertain Solidity exposure (Llama 3.2) against models with confirmed Solidity training but standard context lengths (StarCoder). It also enables evaluating the trade-offs between resource efficiency (1B) and representational capacity (3B) within our practical hardware constraints [Source 565].

3.2 Model Architecture

The model architecture is based on a standard Large Language Model (LLM) framework that incorporates essential components such as tokenization, embedding layers, and a multi-layer transformer stack. In this design, raw Solidity source code is first tokenized into numerical representations and then processed through a series of transformer layers (each blue box in Figure 2, comprising the full self-attention + feed-forward block with its residual connections and layer-norm). This enables the model to capture both local syntactic patterns and long-range dependencies in the code—capabilities that are critical for understanding the complexities inherent in smart contract analysis.

To tailor the generic LLM for smart contract vulnerability detection, the conventional generative output head is replaced with a dedicated classification head. Rather than predicting the next token from an extensive vocabulary, this modified output layer, which is implemented as a multi-layer perceptron (MLP), maps the final (or penultimate) hidden representations directly to a predefined set of vulnerability classes (e.g., reentrancy, access control, safe, vulnerable). Figure 2 illustrates the adapted architecture. This transformation effectively converts the LLM into a traditional classifier pipeline, thereby reducing the risk of generating irrelevant tokens and ensuring that the outputs are confined to the task-specific label set.

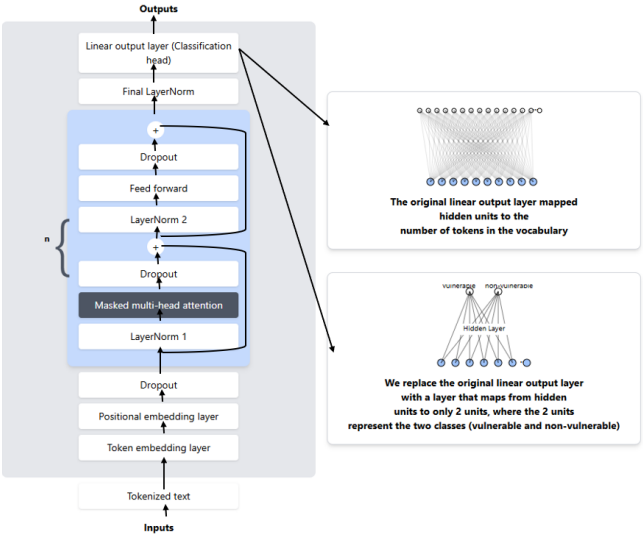


Figure 2: Model architecture for LLM with dedicated classification head

3.3 Datasets

For training our models and conducting the layer-freezing experiments (RQ2), we utilized the ScrawlD dataset developed by Yashavant et al. [38]. ScrawlD is a substantial dataset comprising 9,252 labeled Solidity smart contracts. Vulnerability labels within ScrawlD were determined via a majority-voting consensus among five analysis tools (Osiris, Oyente, Mythril, Slither, SmartCheck), identifying 5,364 contracts with at least one vulnerability according to this methodology.

From this dataset, which originally exhibited an imbalance with roughly two-thirds of contracts labeled as vulnerable, we first partitioned the data into initial training (70%), validation (15%), and test (15%) sets. Recognizing that training on imbalanced data could bias model performance, we addressed this by applying an oversampling technique specifically to the training set. We augmented the training data by replicating instances of the minority class (non-vulnerable contracts) until a balanced distribution between vulnerable and non-vulnerable samples was achieved before the proceeding with the fine-tuning process.

3.4 Evaluation Strategy

Using our stratified data split, we systematically explored the impact of freezing varying numbers of lower transformer layers during fine-tuning for the RQ2 evaluation. For a given model with N total transformer layers we explored configurations across a spectrum defined by the number of frozen bottom layers (k):

(1) **No Frozen Layers ($k = 0$):** All N transformer layers and the classification head were fine-tuned (referred to as Full Fine-Tuning). This served as the performance baseline.
(2) **All Layers Frozen ($k = N$):** Only the classification head was fine-tuned, with all N transformer layers kept frozen (referred to as Head-Only Fine-Tuning). This isolated the head's contribution.
(3) **Intermediate Freezing ($k = 2, 4, 6, ..., < N$):** The bottom k transformer layers were frozen, while the top $N - k$ transformer layers and the classification head were fine-tuned. This allowed us to map the performance impact as the freeze boundary shifted upwards. Table 3 summarizes the different layer-freezing configurations.

Consequently, all these layer-freezing configurations of each model were trained using the balanced training set and subsequently evaluated on the test set derived from ScrawlD.

To address RQ1, which assesses the effectiveness of incorporating a dedicated classification head, we compared our fine-tuned models against the state-of-the-art results published by Ince et al. [17]. To facilitate a direct and equitable comparison, we evaluated our models using the specific test benchmark curated and released by Ince et al. This benchmark test set contains approximately 595 Solidity smart contracts for evaluation purposes. These contracts were labeled using the identical five-tool majority-voting procedure applied to the ScrawlD dataset. To address RQ1, we evaluated our models using the 'Full Fine-Tuning' configuration described above (all N transformer layers and the classification head trained). Their performance on this benchmark was then compared against the results reported by Ince et al. [17] for other models evaluated on the same test set.

Additionally, comparing the performance of these fine-tuned LLMs against non-LLM-based vulnerability detection methods falls outside the scope of this study, which concentrates specifically on efficient LLM adaptation techniques. Such comparisons with traditional static/dynamic analysis or other machine learning approaches have been addressed in prior work [36].

Table 3: Overview of layer-freezing configurations. Each "transformer layer" refers to the blue box (attention + feed-forward) detailed in Figure 2.

Configuration	Trainable Components	Frozen Components
Full Fine-Tuning ($k = 0$)	All N transformer layers & Classification head	None
Head-Only ($k = N$)	Classification head	All N transformer layers
Partial Fine-Tuning ($0 < k < N$)	Top $N - k$ transformer layers & Classification head	Bottom k transformer layers

4 Experiments and Evaluations

4.1 Experimental Setup

4.1.1 Model Variants. For our experiments, we used the four models described in Section 3.1:

- **Llama 3.2 1B**
- **Llama 3.2 3B**
- **StarCoderBase-1b**
- **StarCoderBase-3b**

For both RQ1 and RQ2, the fine-tuning process followed the methodology outlined in Section 3.4. Specifically, for RQ1, each model underwent full fine-tuning with a dedicated classification head. While for RQ2, the fine-tuning process involved applying different layer-freezing conditions, including full fine-tuning, training only the classification head, and various intermediate configurations. This allowed for the creation of trained models tailored to each specific condition.

4.1.2 Resource Used for Fine-Tuning. All experiments were conducted on a single NVIDIA A100 GPU (40 GB VRAM) from the Texas Advanced Computing Center's (TACC) Lonestar6 cluster. For all model fine-tuning, we utilized the LoRA (Low-Rank Adaptation) [14] methodology as our principal approch to fine-tuning the models. For implementing the layer-wise fine-tuning strategy, we leveraged libraries like Ludwig, Hugging Face Transformers and LoRA. This enabled us to selectively freeze specific model layers while applying LoRA adapters to the remaining trainable layers. The complete code implementation, training scripts, datasets, and detailed experimental results are available in our reproducibility package.[1]

4.1.3 Hyperparameter Space. The fine-tuning process was configured with carefully selected hyperparameters to optimize model performance while working within computational constraints. We employed a learning rate of 5e-4, which was empirically determined to provide stable training convergence while avoiding overshooting optimal parameter values. Given the memory constraints of processing lengthy smart contracts, we implemented a batch size of 2 with gradient accumulation steps of 16, effectively simulating a larger batch size of 32 while maintaining manageable memory usage. The models were trained for 5 epochs, which proved sufficient for convergence in our LoRA-based setup. For optimization, we utilized the Paged AdamW optimizer, a memory-efficient variant of Adam that is particularly well-suited for training large language models. This choice was motivated by the need to manage memory usage effectively while maintaining stable training dynamics. Due to computational resource limitations associated with fine-tuning multiple large models across numerous configurations, each experimental condition reported in this study was trained and evaluated once. While this provides valuable performance indicators, we acknowledge that single runs preclude analysis of statistical significance or run-to-run variability.

These hyperparameter choices were guided by both empirical testing and practical constraints, aiming to maximize model performance within the available computational resources while ensuring reliable vulnerability detection capabilities. Importantly, we applied the same hyperparameter configuration across all model and layer-freezing conditions to ensure that any observed performance differences stem solely from the layer-freezing strategy rather than model-specific tuning.

4.2 Results and Discussion

RQ1: Effectiveness of Classification Head Approach. To address our first research question on the effectiveness of fine-tuning LLMs with a dedicated classifier head for smart contract vulnerability detection, we conducted a comprehensive evaluation comparing our approach with state-of-the-art models. Table 4 presents the performance metrics across various models for binary classification. This evaluation focuses on binary classification across key metrics: precision, recall, F_1 score, and accuracy.

Our fine-tuned models demonstrated substantial improvements over the baseline approaches reported in Ince et al. [17]. Among our models, Llama 3.2 3B-AdapT achieved the highest performance with 77.5% accuracy, 76.5% precision, 77.8% recall, and an F1 score of 0.771. This places it on par with the top-performing GPT-3.5FT model (77.6% accuracy), despite the latter likely having significantly more parameters. The comparable performance indicates that our classification head approach effectively adapts LLM for this specialized domain task.

A clear trend emerges when examining performance across model sizes within the same architecture family. The 3B parameter variants consistently outperformed their 1B counterparts across all metrics. For instance, Llama 3.2 3B-AdapT achieved 3.5 percentage points higher accuracy than Llama 3.2 1B-AdapT (77.5% versus 74.0%), while StarCoder 3B-AdapT similarly outperformed StarCoder 1B-AdapT by 6.3 percentage points (68.8% versus 62.5%). This pattern suggests that increased model capacity contributed to better discrimination of vulnerability patterns, though the performance gains appeared to scale sub-linearly with parameter count.

Architecture differences also played a significant role in performance. The Llama 3.2 models consistently outperformed their StarCoder counterparts of equivalent size. This performance gap (approximately 8.7 percentage points for 3B models and 11.5 percentage points for 1B models) indicated that architectural design choices

[1] https://bit.ly/paper_repo_adapT_LLM

Table 4: Comparison of Vulnerability Detection Performance Across Models for Binary Classification. Models marked with * are from this paper; results and descriptions for other models are from Ince et al. [17].

Model	Prec.	Rec.	F_1	Acc.
Llama 3.2 3B-AdapT* *Llama 3.2 3B fine-tuned w/ classif. head*	0.765	0.778	0.771	0.775
Llama 3.2 1B-AdapT* *Llama 3.2 1B fine-tuned w/ classif. head*	0.742	0.735	0.738	0.740
StarCoder 3B-AdapT* *StarCoder 3B fine-tuned w/ classif. head*	0.685	0.682	0.683	0.688
StarCoder 1B-AdapT* *StarCoder 1B fine-tuned w/ classif. head*	0.628	0.610	0.619	0.625
GPT-3.5FT *OpenAI GPT-3.5 Turbo fine-tuned (Ince)*	0.770	0.782	0.776	0.776
GPT-4 *OpenAI GPT-4 zero-shot eval (Ince)*	0.675	0.646	0.660	0.661
DL-Instruct *34B Code Llama Instruct fine-tuned (Ince)*	0.774	0.443	0.563	0.648
GPTLens *GPTLens method (GPT-4 Turbo) eval (Ince)*	0.533	0.988	0.692	0.564
DL-Foundation *34B Code Llama Foundation fine-tuned (Ince)*	0.517	0.993	0.680	0.521

and pre-training strategies substantially influenced a model's adaptability to security-focused code analysis tasks, potentially more so than raw parameter count alone. When comparing our models to other approaches in the field, we observed distinct trade-off patterns. Models like GPTLens and DL-Foundation exhibited extremely high recall (98.8% and 99.3% respectively) but suffered from low precision (53.3% and 51.7%), resulting in lower overall accuracy. This suggested these models were optimized for maximizing vulnerability coverage at the expense of generating numerous false positives. In contrast, our fine-tuned models and GPT-3.5FT achieved more balanced precision and recall scores, indicating better overall discrimination capability.

The distribution of performance metrics across all evaluated models revealed an important insight: achieving high precision in smart contract vulnerability detection remained challenging. Even the best-performing models achieved precision in the mid-to-high 70% range, suggesting inherent difficulty in definitively identifying code as vulnerable without false positives. This challenge likely stemmed from the subtle nature of many vulnerabilities, where small variations in context or implementation could determine whether a pattern represented a genuine security risk.

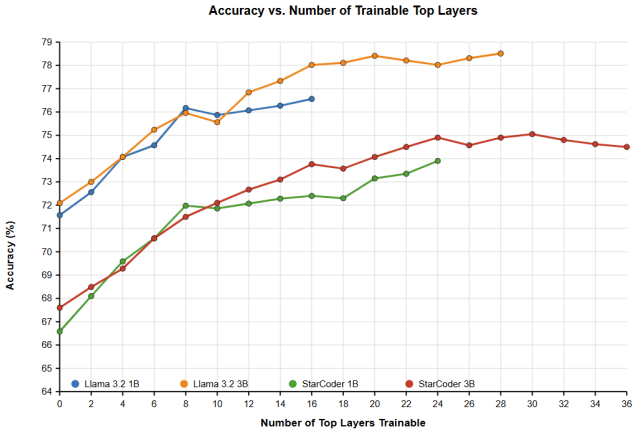


Figure 3: Impact of layer freezing on model performance metrics across four models. The x-axis represents the number of trainable top layers, while the y-axis shows the corresponding accuracy

Our Llama 3.2 3B-AdapT model notably outperformed GPT-4 in this specific task, with accuracy of 77.5% versus 66.1%. This outcome was particularly noteworthy given GPT-4's substantially larger size and broader capabilities, highlighting that task-specific fine-tuning with an appropriate architecture could exceed the performance of more general, larger models on specialized tasks like vulnerability detection.

These results collectively demonstrated that fine-tuning moderate-sized LLMs with dedicated classification heads offered an effective approach for smart contract vulnerability detection. The classification head architecture appeared to successfully leverage the rich contextual representations of transformer-based models while constraining outputs to a well-defined set of vulnerability classes, avoiding potential ambiguities in token-generation approaches. Furthermore, the performance achieved by our Llama 3.2 3B-AdapT model indicated that this approach could yield state-of-the-art results while remaining computationally more efficient than employing the largest available models.

RQ2: Impact of Layer-Specific Fine-tuning. To investigate how selective layer freezing affected model performance and computational efficiency, we conducted experiments across four models (Llama 3.2 1B, Llama 3.2 3B, StarCoder 1B, and StarCoder 3B), systematically freezing increasing numbers of bottom layers while measuring performance and resource usage.

Our analysis revealed a compelling pattern of diminishing returns as lower layers were frozen. Both accuracy and F1 scores remained quite stable even when a significant portion of bottom layers were prevented from updating during fine-tuning. For instance, freezing half of the layers in most models resulted in performance drops of less than 1.7 percentage points compared to full fine-tuning. This consistent pattern across all tested architectures, as seen in Figures 3 and 4, suggests that substantial task-specific adaptation occurs in the upper layers. With such selective layer freezing strategies, training time decreased by approximately 10-20% and memory usage also decreased by about 10-20%, as illustrated in Figure 5. For

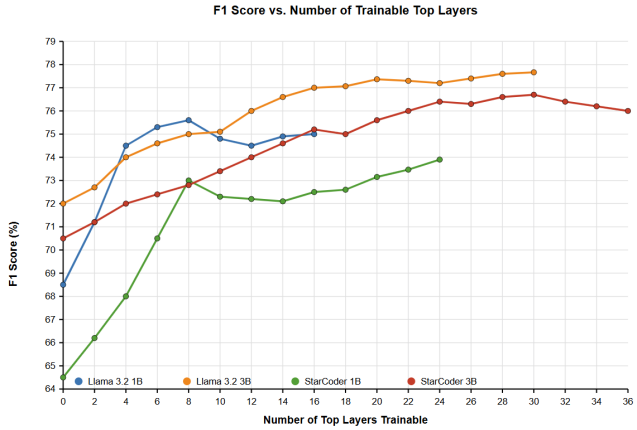


Figure 4: Impact of layer freezing on model performance metrics across four models. The x-axis represents the number of trainable top layers, while the y-axis shows the corresponding F1 Score

resource-constrained environments, these findings demonstrated a viable path to deploying effective LLMs that would otherwise be computationally prohibitive.

Model size also influenced freeze-resilience. The larger Llama 3.2 3 B retained ≥ 75% accuracy with just the top six layers trainable (≈ 20% of the network), whereas the 1 B variant slipped below that mark once more than ~80% of its layers were frozen. This indicates that the extra capacity in the 3 B model provides enough representational redundancy to tolerate more aggressive parameter-efficiency strategies.

These results has important implications for practical applications of LLM fine-tuning. By selectively fine-tuning only the top layers, practitioners could achieve near-optimal performance while reducing computational requirements. In the context of smart contract vulnerability detection, our findings suggested that the general code understanding capabilities encoded during pre-training remained largely applicable without modification, while task-specific adaptation for security vulnerabilities could be effectively captured through updates to the upper layers alone. This supported the principle that recognizing vulnerability patterns relied more on high-level semantic understanding than low-level syntactic processing.

In summary, our experimental results demonstrated that restricting fine-tuning to specific high-level Transformer layers while freezing lower layers achieved competitive performance with significantly reduced computational requirements, making LLM-based vulnerability detection more practical for real-world applications.

5 Related Work

Research in smart contract vulnerability detection encompassed diverse methodological approaches that collectively aimed to address the critical challenge of securing blockchain-based programs. The landscape of vulnerability detection techniques spanned from

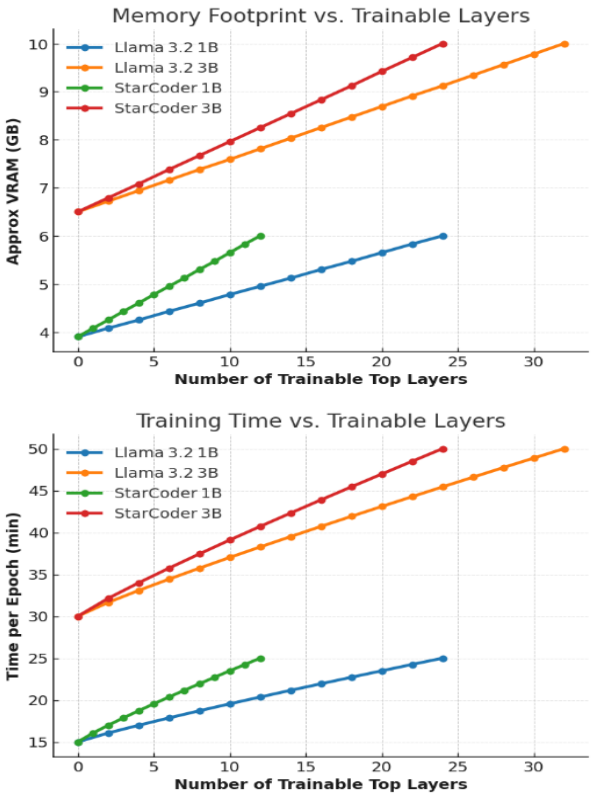


Figure 5: Computational resource usage versus the number of trainable top layers during fine-tuning. Top: Approximate VRAM footprint (GB). Bottom: Training time per epoch (minutes). Results shown for Llama 3.2 (1B, 3B) and StarCoder (1B, 3B) models.

traditional software engineering methods, such as static and dynamic analysis, to machine learning-based approaches that leveraged pattern recognition and automated analysis. In the domain of machine learning-based approaches, applications of LLMs had also been actively pursued in recent times. This section systematically reviewed recent advancements in using LLMs specifically for enhancing smart contract vulnerability detection.

Several studies presented robust LLM-based frameworks that enhanced vulnerability detection over traditional security tools. LLM-SmartAudit, for instance, introduced a multi-agent LLM framework that captured complex contract interactions, allowing the identification of previously undetected logic vulnerabilities through a collaborative LLM agent system [35]. Similarly, FELLMVP employed an ensemble of fine-tuned LLMs to classify multiple vulnerability types, achieving higher accuracy and efficiency than both static and dynamic analysis methods [21]. Additionally, LLMSmartSec integrated LLMs with annotated control flow graphs to improve vulnerability detection and remediation, showcasing the potential of hybrid approaches in smart contract security [23]. Although these approaches offered effective strategies, they still relied on autoregressive generation, inheriting the limitations of LLM.

Despite their advancements, LLM-based methods faced certain challenges. False positives remained a persistent issue; GPTLENS employed adversarial techniques that segmented generation and validation stages to mitigate incorrect detections, underscoring the need for refining detection accuracy in LLM systems [16]. Targeted LLM applications also showed promise in detecting specific logic-related vulnerabilities within smart contracts. Soley and GPTScan, for instance, leveraged fine-tuned LLMs to identify intricate logic vulnerabilities that static analysis tools often overlooked, such as misaligned state changes and errors in business logic [28]. By focusing on Solidity code intricacies, these models demonstrated the potential of LLMs in detecting nuanced vulnerabilities specific to contract logic.

Additionally, Context-Driven Prompting enhanced detection precision by structuring prompts to emphasize relevant portions of the code, thereby optimizing the LLM's focus during analysis [4, 30]. RAG-Enhanced LLMs incorporated Retrieval-Augmented Generation (RAG), which retrieved relevant contextual information during detection, thereby improving precision by extending the LLM's ability to generalize and adapt to complex vulnerability patterns [40]. These studies indicated that improved prompting could significantly boost LLM-based detection in both accuracy and efficiency.

Finally, the application of LLMs to multiple blockchain platforms remained an area with unrealized potential. LLM-based vulnerability detection methods are significantly advancing smart contract security, addressing challenges through novel frameworks, prompting strategies, and hybrid techniques as reviewed above. However, to the best of our knowledge, the specific adaptation strategies investigated in this paper—evaluating the effectiveness of a dedicated classification head compared to standard token generation (RQ1) and systematically analyzing the performance and efficiency trade-offs of layer-wise fine-tuning by freezing lower layers (RQ2)—have not been the primary focus of prior work in this domain. Consequently, challenges such as false positives, computational costs, and cross-platform adaptability persist. Continued research in optimizing LLM frameworks, including efficient adaptation techniques like those studied here, expanding platform applicability, and improving resource efficiency will be vital to fully realizing the potential of LLMs in the evolving landscape of blockchain security.

6 Conclusion

In this study, we investigated two core questions: first, whether a specialized classification head could enhance the effectiveness of Large Language Models (LLMs) in identifying smart contract vulnerabilities; and second, how varying the number of frozen lower layers during fine-tuning impacts resource consumption and performance. Our experiments demonstrated that fine-tuning with dedicated classification heads yielded competitive results. Notably, our Llama 3.2 3B-AdapT model reached 77.5% accuracy, performing comparably to the fine-tuned GPT-3.5FT model despite potential differences in underlying architecture and size. This highlights the effectiveness of the classification head approach for this task. Furthermore, our investigation into layer freezing revealed that performance remained stable even when a substantial number of lower transformer layers were frozen, resulting in minimal degradation while offering notable efficiency gains. We observed reductions in both training time and memory usage of approximately 10-20% when fine-tuning only the upper layers, making LLM adaptation more feasible under resource constraints. Larger models exhibited greater resilience; for example, Llama 3.2 3B-AdapT maintained high accuracy even when a large proportion of the lower layers were frozen (leaving only upper layers trainable), suggesting that general code understanding resides largely in lower layers while vulnerability detection patterns are primarily learned in upper layers.

Building on these findings, our future research will pursue several promising directions. First, extending our classification approach to multi-class vulnerability detection would enhance practical utility by providing specific vulnerability classifications rather than binary judgments. Second, exploring the application of this fine-tuning scheme to other blockchain platforms beyond Ethereum would test the generalizability of our approach across diverse programming languages and execution models. Finally, implementing adaptive layer-freezing schedules—where the number of trainable layers was dynamically adjusted based on validation metrics—might further optimize computational efficiency. These directions collectively promise to advance the accessibility and effectiveness of LLM-based vulnerability detection systems for blockchain security applications, potentially making sophisticated security analysis tools available even in resource constrained environments.

Acknowledgments

Lei's work is partly supported by a research grant (70NANB21H092) from Information Technology Lab of National Institute of Standards and Technology (NIST). Ji's work is supported in part by National Science Foundation grants (2516003).

The views, opinions, and/or findings expressed in this material are those of the authors and should not be interpreted as representing the official views of the National Science Foundation, National Institute of Standards and Technology, or the U.S. Government.

The authors acknowledge the Texas Advanced Computing Center (TACC) at The University of Texas at Austin for providing computational resources that have contributed to the research results reported within this paper. URL: http://www.tacc.utexas.edu

References

[1] Md Tauseef Alam, Raju Halder, and Abyayananda Maiti. 2024. Detection Made Easy: Potentials of Large Language Models for Solidity Vulnerabilities. doi:10.48550/arXiv.2409.10574 arXiv:2409.10574 [cs].

[2] Andry Alamsyah, Gede Natha Wijaya Kusuma, and Dian Puteri Ramadhani. 2024. A Review on Decentralized Finance Ecosystems. *Future Internet* 16, 3 (Feb. 2024), 76. doi:10.3390/fi16030076

[3] Emily M. Bender, Timnit Gebru, Angelina McMillan-Major, and Shmargaret Shmitchell. 2021. On the Dangers of Stochastic Parrots: Can Language Models Be Too Big?. In *Proceedings of the 2021 ACM Conference on Fairness, Accountability, and Transparency* (Virtual Event, Canada) *(FAccT Â21)*. Association for Computing Machinery, New York, NY, USA, 610–623. doi:10.1145/3442188.3445922

[4] Mohamed Salah Bouafif, Chen Zheng, Ilham Ahmed Qasse, Ed Zulkoski, Mohammad Hamdaqa, and Foutse Khomh. 2024. A Context-Driven Approach for Co-Auditing Smart Contracts with The Support of GPT-4 code interpreter. arXiv:2406.18075 [cs.SE] https://arxiv.org/abs/2406.18075

[5] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya

Sutskever, and Dario Amodei. 2020. Language Models Are Few-Shot Learners. doi:10.48550/arXiv.2005.14165 arXiv:2005.14165 [cs]

[6] CertiK. 2023. CertiK - Hack3d: The Web3 Security Report 2023. https://certik.com/resources/blog/hack3d-the-web3-security-report-2023. Accessed: 2024-09-02.

[7] Chong Chen, Jianzhong Su, Jiachi Chen, Yanlin Wang, Tingting Bi, Jianxing Yu, Yanli Wang, Xingwei Lin, Ting Chen, and Zibin Zheng. 2024. When ChatGPT Meets Smart Contract Vulnerability Detection: How Far Are We? *ACM Transactions on Software Engineering and Methodology* (Nov. 2024), 3702973. doi:10.1145/3702973

[8] Isaac David, Liyi Zhou, Kaihua Qin, Dawn Song, Lorenzo Cavallaro, and Arthur Gervais. 2023. Do you still need a manual smart contract audit? doi:10.48550/arXiv.2306.12338 arXiv:2306.12338 [cs].

[9] Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, and Luke Zettlemoyer. 2023. QLoRA: Efficient Finetuning of Quantized LLMs. https://arxiv.org/abs/2305.14314. arXiv:2305.14314 [cs.LG] https://arxiv.org/abs/2305.14314

[10] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. arXiv:1810.04805 [cs.CL] https://arxiv.org/abs/1810.04805

[11] Suchin Gururangan, Ana Marasović, Swabha Swayamdipta, Kyle Lo, Iz Beltagy, Doug Downey, and Noah A. Smith. 2020. Don't Stop Pretraining: Adapt Language Models to Domains and Tasks. arXiv:2004.10964 [cs.CL] https://arxiv.org/abs/2004.10964

[12] Arthur Hemmer, Mickaël Coustaty, Nicola Bartolo, Jérôme Brachat, and Jean-Marc Ogier. 2023. Lazy-k: Decoding for Constrained Token Classification. arXiv:2312.03367 [cs.CL] https://arxiv.org/abs/2312.03367

[13] Jeremy Howard and Sebastian Ruder. 2018. Universal Language Model Fine-tuning for Text Classification. arXiv:1801.06146 [cs.CL] https://arxiv.org/abs/1801.06146

[14] Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. 2021. LoRA: Low-Rank Adaptation of Large Language Models. arXiv:2106.09685 [cs] http://arxiv.org/abs/2106.09685

[15] Sihao Hu, Tiansheng Huang, Fatih İlhan, Selim Furkan Tekin, and Ling Liu. 2023. 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)*. 297–306. doi:10.1109/TPS-ISA58951.2023.00044

[16] Sihao Hu, Tiansheng Huang, Fatih İlhan, Selim Furkan Tekin, and Ling Liu. 2023. Large Language Model-Powered Smart Contract Vulnerability Detection: New Perspectives. arXiv:2310.01152 [cs.CR] https://arxiv.org/abs/2310.01152

[17] Peter Ince, Xiapu Luo, Jiangshan Yu, Joseph K. Liu, and Xiaoning Du. 2024. *Detect Llama - Finding Vulnerabilities in Smart Contracts Using Large Language Models*. Springer Nature Singapore, 424–443. doi:10.1007/978-981-97-5101-3_23

[18] Satpal Singh Kushwaha, Sandeep Joshi, Dilbag Singh, Manjit Kaur, and Heung-No Lee. 2022. Systematic Review of Security Vulnerabilities in Ethereum Blockchain Smart Contract. *IEEE Access* 10 (2022), 6605–6621. doi:10.1109/ACCESS.2021.3140091

[19] Jaejun Lee, Raphael Tang, and Jimmy Lin. 2019. What Would Elsa Do? Freezing Layers During Transformer Fine-Tuning. arXiv:1911.03090 [cs.CL] https://arxiv.org/abs/1911.03090

[20] Zongxi Li, Xianming Li, Yuzhang Liu, Haoran Xie, Jing Li, Fu lee Wang, Qing Li, and Xiaoqin Zhong. 2023. Label Supervised LLaMA Finetuning. arXiv:2310.01208 [cs.CL] https://arxiv.org/abs/2310.01208

[21] Yu Luo, Weifeng Xu, Karl Andersson, Mohammad Shahadat Hossain, and Dianxiang Xu. 2024. FELLMVP: An Ensemble LLM Framework for Classifying Smart Contract Vulnerabilities. In *2024 IEEE International Conference on Blockchain (Blockchain)*. 89–96. doi:10.1109/Blockchain62396.2024.00021

[22] Marius Mosbach, Maksym Andriushchenko, and Dietrich Klakow. 2021. On the Stability of Fine-tuning BERT: Misconceptions, Explanations, and Strong Baselines. arXiv:2006.04884 [cs.LG] https://arxiv.org/abs/2006.04884

[23] Viraaji Mothukuri, Reza M. Parizi, and James L. Massa. 2024. LLMSmartSec: Smart Contract Security Auditing with LLM and Annotated Control Flow Graph. In *2024 IEEE International Conference on Blockchain (Blockchain)*. 434–441. doi:10.1109/Blockchain62396.2024.00064

[24] Satoshi Nakamoto. 2008. Bitcoin: A peer-to-peer electronic cash system. *Satoshi Nakamoto* (2008).

[25] Matthew E. Peters, Sebastian Ruder, and Noah A. Smith. 2019. To Tune or Not to Tune? Adapting Pretrained Representations to Diverse Tasks. arXiv:1903.05987 [cs.CL] https://arxiv.org/abs/1903.05987

[26] Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. 2019. Language Models are Unsupervised Multitask Learners. https://api.semanticscholar.org/CorpusID:160025533

[27] Heidelinde Rameder, Monika di Angelo, and Gernot Salzer. 2022. Review of Automated Vulnerability Analysis of Smart Contracts on Ethereum. *Frontiers in Blockchain* 5 (March 2022). doi:10.3389/fbloc.2022.814977

[28] Majd Soud, Waltteri Nuutinen, and Grischa Liebel. 2024. Soley: Identification and Automated Detection of Logic Vulnerabilities in Ethereum Smart Contracts Using Large Language Models. arXiv:2406.16244 [cs.ET] https://arxiv.org/abs/2406.16244

[29] André Storhaug, Jingyue Li, and Tianyuan Hu. 2023. Efficient Avoidance of Vulnerabilities in Auto-completed Smart Contract Code Using Vulnerability-constrained Decoding. In *2023 IEEE 34th International Symposium on Software Reliability Engineering (ISSRE)*. 683–693. doi:10.1109/ISSRE59848.2023.00035

[30] Yuqiang Sun and Ying Liu. 2023. GPTScan: Detecting Logic Vulnerabilities in Smart Contracts by Combining GPT with Program Analysis. In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE)*.

[31] Xueyan Tang, Yuying Du, Alan Lai, Ze Zhang, and Lingzhi Shi. 2023. Deep Learning-Based Solution for Smart Contract Vulnerabilities Detection. *Scientific Reports* 13, 1 (Nov. 2023), 20106. doi:10.1038/s41598-023-47219-0

[32] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2023. Attention Is All You Need. doi:10.48550/arXiv.1706.03762 arXiv:1706.03762 [cs]

[33] Qiping Wei, Fadul Sikder, Huadong Feng, Yu Lei, Raghu Kacker, and Richard Kuhn. 2023. SmartExecutor: Coverage-Driven Symbolic Execution Guided by a Function Dependency Graph. In *2023 5th Conference on Blockchain Research & Applications for Innovative Networks and Services (BRAINS)*. 1–8. doi:10.1109/BRAINS59668.2023.10316942

[34] Qiping Wei, Fadul Sikder, Huadong Feng, Yu Lei, Raghu Kacker, and Richard Kuhn. 2025. SmartExecutor: Coverage-Driven Symbolic Execution Guided via State Prioritization and Function Selection. *Distributed Ledger Technologies: Research and Practice* 4, 1 (March 2025), 1–29. doi:10.1145/3678188

[35] Zhiyuan Wei, Jing Sun, Zijiang Zhang, and Xianhao Zhang. 2024. LLM-SmartAudit: Advanced Smart Contract Vulnerability Detection. arXiv:2410.09381 [cs.CR] https://arxiv.org/abs/2410.09381

[36] Zhiyuan Wei, Jing Sun, Zijian Zhang, Xianhao Zhang, Meng Li, and Mauro Conti. 2025. FTSmartAudit: A Knowledge Distillation-Enhanced Framework for Automated Smart Contract Auditing Using Fine-Tuned LLMs. arXiv:2410.13918 [cs.CR] https://arxiv.org/abs/2410.13918

[37] Huaiguang Wu, Yibo Peng, Yaqiong He, and Jinlin Fan. 2024. A Review of Deep Learning-Based Vulnerability Detection Tools for Ethernet Smart Contracts. *Computer Modeling in Engineering & Sciences* 140, 1 (2024), 77–108. doi:10.32604/cmes.2024.046758

[38] Chavhan Sujeet Yashavant, Saurabh Kumar, and Amey Karkare. 2022. ScrawlD: A Dataset of Real World Ethereum Smart Contracts Labelled with Vulnerabilities. arXiv:2202.11409 [cs.CR] https://arxiv.org/abs/2202.11409

[39] Jason Yosinski, Jeff Clune, Yoshua Bengio, and Hod Lipson. 2014. How transferable are features in deep neural networks? arXiv:1411.1792 [cs.LG] https://arxiv.org/abs/1411.1792

[40] Jeffy Yu. 2024. Retrieval Augmented Generation Integrated Large Language Models in Smart Contract Vulnerability Detection. doi:10.48550/arXiv.2407.14838

[41] Lei Yu, Shiqi Chen, Hang Yuan, Peng Wang, Zhirong Huang, Jingyuan Zhang, Chenjie Shen, Fengjun Zhang, Li Yang, and Jiajia Ma. 2024. Smart-LLaMA: Two-Stage Post-Training of Large Language Models for Smart Contract Vulnerability Detection and Explanation. doi:10.48550/arXiv.2411.06221 arXiv:2411.06221 [cs].

[42] Zibin Zheng, Shaoan Xie, Hongning Dai, Xiangping Chen, and Huaimin Wang. 2017. An Overview of Blockchain Technology: Architecture, Consensus, and Future Trends. In *2017 IEEE International Congress on Big Data (BigData Congress)*. 557–564. doi:10.1109/BigDataCongress.2017.85