

Large Language Model based Smart Contract Auditing with LLMBugScanner

Yining Yuan*, Yifei Wang*, Yichang Xu,
Zachary Yahn, Sihao Hu, and Ling Liu **

School of Computer Science, Georgia Institute of Technology,
Atlanta, GA 30332, United States

{yyuan394, ywang4343, xuyichang, zachary.yahn, sihaohu, 1172}@gatech.edu

Abstract. This paper presents LLMBugScanner – a large language model (LLM) powered approach to smart contract vulnerability detection with fine-tuning and ensemble learning. First, the paper identifies the challenges of leveraging for auditing and bug detection of smart contract programs. For example, different pre-trained LLMs may have different reasoning capabilities, and a single LLM may not be consistently effective for all vulnerability types and/or all type of smart contract programs. Such problems also persist across fine-tuned LLMs. To address such challenges, this article explores the adaptation of domain knowledge and ensemble learning and integrates these two synergistic strategies for effective and enhanced generalization performance of smart contract vulnerability detection. With domain knowledge adaptation, we fine-tune LLMs on complementary datasets to provide both general code interpretation and instructional supervision, using parameter-efficient techniques to reduce computational cost. With LLM-ensemble, we capitalize on the complimentary wisdom of diverse LLMs to generate improved reasoning results through consensus based conflict resolution. Extensive experiments are performed on multiple popular LLMs. We compare LLMBugScanner with both individual pre-trained LLMs and their fine-tuned versions in terms of their smart contract vulnerability detection performance. Our measurement results show that LLMBugScanner delivers consistent accuracy gains and better generalization, highlighting LLMBugScanner as a principled, cost-effective, and extensible framework for smart contract auditing.

Keywords: Large language model, LLM finetuning, smart contract, vulnerability detection, LLM-Ensemble

1 Introduction

Smart contracts are self-executing programs that are managed and executed on the blockchain, enabling decentralized applications such as DeFi and NFTs to

* These authors contributed equally to this work.

** Corresponding author: Ling Liu

operate without intermediaries, collectively managing digital assets [26]. This immutability (the contract code cannot be directly modified once deployed) serves as a foundation for trust and transparency in decentralized systems. However, it also faces critical risk: Even small bugs in smart contracts can cause catastrophic consequences, including theft, token devaluation, or permanent/unauthorized locking of funds. Research analysis confirms that common programming pitfalls in Ethereum contracts create exploitable security risks that can enable attackers to steal assets or disrupt functionality [22,41]. A quintessential example is the 2016 The DAO exploit, where a reentrancy bug allowed attackers to siphon off about \$60 million worth of Ether, resulting in a controversial hard fork of the Ethereum blockchain [31]. More recently, DeFi platforms have seen security breaches on an even larger scale, underscoring the immense financial risks involved [20,17]. As a result, there's growing demand for reliable detection tools that can identify vulnerabilities before deployment. A systematic literature review identified nearly 192 different types of Ethereum smart contract vulnerabilities and over 200 detection tools [21].

Traditional static and dynamic program analyzers [7,15,40,32] form the core of current auditing pipelines. However, prior empirical studies reveal persistent weaknesses, including high rates of false-positives, incomplete coverage of complex control flows, function-specific blind spots, and limited robustness to new vulnerability patterns [22,41]. These limitations arise because rule-based analyzers depend on pre-defined patterns and fail to capture the actual logic or intent expressed in the code [18].

Recent advances in large language models (LLMs) present an opportunity for developing a new paradigm for vulnerability detection [3]. We argue that LLMs offer key advantages over static analyzers through their generality, contextual reasoning, and interpretability. For example, studies show that LLMs trained on code corpora achieve superior semantic code understanding and generation [37]. Unlike rule-based tools, LLMs holds the potential to detect both known and new vulnerabilities, by reasoning across different semantics of smart contract programs, from logic, comments, to specifications, reducing false negatives in complex issues, e.g., reentrancy or access control flaws. Moreover, LLMs can explain their reasoning in natural language, providing interpretable insights that help auditors and developers validate and prioritize bug fixes. Several projects [18,39] have pioneered the use of LLMs as foundational components in smart contract auditing frameworks. Despite these advances, existing LLM-based approaches remain limited. First, single-model fine-tuning can overfit to specific vulnerability types or datasets, leading to inconsistent predictions across domains [13]. Second, different LLMs exhibit different reasoning capabilities and no single LLM or its fine-tuned version can correctly and persistently outperform others for smart contract vulnerability detection [18].

To address these issues, we present LLMBugScanner, an LLM-based smart contract vulnerability detection framework with parameter-efficient adaptation. The design of LLMBugScanner explores the Synergism of domain knowledge adaptation and LLM-ensemble to improve its generalization performance. First,

we incorporate *Domain knowledge adaptation* with heterogeneous smart contract benchmark datasets and parameter-efficient optimization [16]. This allows LLMBugScanner to capture diverse semantic patterns of smart contract programs, while keeping fine-tuning of LLMs resource-efficient. Second, we develop an *LLM-Ensemble* approach to combine the reasoning results of multiple independently fine-tuned LLMs to fortify the robustness of the vulnerability detection of LLMBugScanner across diverse vulnerability categories. We performed experiments on 108 real-world smart contracts, all of which were reported to contain vulnerabilities in the Common Vulnerabilities and Exposures (CVEs) database [8]. Empirically, LLMBugScanner achieves a 60% top 5 detection accuracy on 108 real-world vulnerable contracts from the curated subset of CVE-labeled Solidity contracts[18], outperforming single-model baselines by 19% while maintaining efficiency. The results highlight LLMBugScanner as a scalable and robust approach for smart contract vulnerability detection.

2 Background and Related Work

2.1 Smart Contract Basics

Smart contracts are self-executing programs stored on blockchain platforms (e.g., Ethereum), designed to enforce agreements without intermediaries [26]. Once deployed, the contract code is immutable and publicly accessible, providing transparency but also meaning any vulnerabilities cannot be patched in place [33]. This immutability, combined with the high financial stakes, makes smart contract flaws particularly critical [35]. Unfortunately, real-world smart contracts have been riddled with bugs and security vulnerabilities. Researchers have documented a broad taxonomy of common smart contract vulnerabilities from reentrancy bugs to integer overflows, that attackers can exploit to steal funds or disrupt contract logic [30,4]. To contextualize these risks, Table 1 categorizes common smart contract vulnerabilities across different layers of the Ethereum ecosystem, offering a structured overview that highlights the diverse sources of attack surfaces and motivates the need for automated vulnerability detection.

Several tools and frameworks now exist to detect known vulnerability patterns in smart contracts. Early security analyzers like Oyente showed that automated symbolic analysis could find prevalent bugs such as reentrancy and timestamp dependence [30]. Subsequent static analyzers formalized vulnerability patterns as compliance and violation conditions to flag insecure code constructs [40]. These analyzers successfully identified many flawed contracts in the wild. For instance, Nikolić et al. scanned roughly one million Ethereum contracts and uncovered tens of thousands of vulnerable instances of greedy or suicidal contracts that lock funds or destroy ownership improperly [33]. The cumulative lesson from this line of work is that smart contracts are prone to recurring bug types despite their simplicity, and pattern-based detectors can catch certain classes of bugs [33]. However, traditional approaches rely heavily on expert-crafted rules and formal properties. This reliance yields a fundamental limitation: when contracts deviate from known patterns or when new vulnerability variants emerge,

Table 1: **Categorization of smart contract vulnerabilities.** Adapted from Atzei et al. [4] and extended with insights from later works [30,48]. Vulnerabilities are grouped by layer (Solidity language, the Ethereum Virtual Machine (EVM), and blockchain design) to provide a holistic view of common pitfalls and attack surfaces in Ethereum smart contracts.

| Layer | Vulnerability | Description |
|---------------------------------------|---|---|
| Solidity programming language | | |
| | Denial of Service (block gas limit) | Transactions fail if gas usage exceeds the block gas limit (e.g., large loops or arrays). |
| | Denial of Service (failed call) | Repeated failing external calls in loops can block contract progress. |
| | Randomness via <code>blockhash</code> | <code>blockhash</code> is predictable and miner-influenced; unsafe for randomness. |
| | <code>tx.origin</code> misuse | Using <code>tx.origin</code> for authentication enables phishing-style attacks. |
| | Integer over/underflow | Arithmetic beyond type bounds can corrupt contract state. |
| | Re-entrancy | External calls before state updates enable repeated withdrawals (e.g., DAO attack). |
| | Mishandled exceptions | Failing to propagate or handle call errors creates hidden faults. |
| | Gasless <code>send</code> | Fallbacks exceeding the 2300 gas stipend fail, potentially exploitable. |
| | Gas-costly patterns | Inefficient loops or storage use waste gas (e.g., flagged by GASPER). |
| | Call to unknown | Fallback invoked on undefined functions or raw Ether transfers. |
| | Hash collision <code>abi.encodePacked</code> | Using <code>abi.encodePacked()</code> with variable-length arguments (e.g., strings) can produce identical byte sequences for different inputs (e.g., "abc","def" vs. "ab","cde"), leading to hash or signature collisions in verification logic. |
| | Insufficient gas griefing | Starving subcalls of gas forces upstream reverts. |
| | Unprotected Ether withdrawal | Missing access control permits unauthorized withdrawals. |
| | Floating pragma | Unpinned compiler version risks inconsistent builds or compiler bugs. |
| | Default visibility | Omitted visibility exposes unintended public functions. |
| | <code>DELEGATECALL</code> misuse | Executes foreign code in caller context, risking state contamination. |
| | Unprotected <code>selfdestruct</code> | Allows deletion or asset loss without proper authorization. |
| Ethereum Virtual Machine (EVM) | | |
| | Immutable bugs | Deployed bytecode is immutable; defects are permanent. |
| | Ether lost in transfer | Sending to non-code or orphan addresses irreversibly loses funds. |
| Blockchain design layer | | |
| | Timestamp dependency | Miners can bias timestamps to influence contract logic. |
| | Transaction-ordering dependency | Ordering/front-running exploits race conditions in the mempool. |
| | Lack of transactional privacy | Public on-chain data can be inferred and misused. |
| | Untrusted oracles | Malicious or erroneous off-chain sources can corrupt on-chain state. |

purely rule-based tools struggle to keep up [35]. In practice, popular analyzers like Mythril or Slither report high false positive rates and often miss nuanced flaws that do not match their predefined signatures [14]. The need for more flexible, learning-based analysis techniques has become apparent as the complexity and volume of smart contracts grow [35].

2.2 LLM-based Vulnerability Detection

A key factor driving the paradigm shift from traditional approaches [29,51] in software vulnerability detection is the proven ability of code-oriented LLMs to effectively model both the semantic meaning and structural patterns of source code [37]. However, early evaluations show that generic LLMs can sometimes flag vulnerability patterns, frequently produce false positives [9], resulting in a low precision and necessitate exhausting manual verification efforts [18]. To mitigate the above-mentioned issues, multi-agent frameworks that enable LLMs to assume different roles have been introduced [19]. A representative example in smart contract vulnerability detection is GPTLens [18], which decomposes the detection task into two stages with distinct agent responsibilities: auditors that propose potential vulnerabilities, and a critic that refines the outputs and helps reduce false positive rates [18]. Later, LLM-SmartAudit employs a multi-agent conversational structure: individual agents specialize (e.g. code reader, logic checker), cross-validate each other’s outputs, and jointly produce a final audit report. Experiments show significant improvements over traditional tools [43]. Smartify extends this approach across multiple smart-contract languages (Solidity and Move) by deploying an ensemble of fine-tuned agents specialized in detection and repair tasks, outperforming several standalone LLMs [25]. Recent surveys of LLM-based multi-agent systems also highlight that such collaborative approaches can reduce hallucination, offer richer reasoning paths, and mimic human auditing workflows more closely [28]. These multi-agent approaches open a new frontier: rather than relying on one monolithic LLM, we can compose multiple specialized agents or models that reason and critique each other, potentially improving robustness and interpretability in contract auditing.

3 Motivation

3.1 Domain Knowledge Adaptation

Domain-specific fine-tuning has become a central strategy in adapting pre-trained LLMs to specialized downstream tasks. It enables a model to transfer general code understanding ability and programming knowledge acquired during pre-training to specific domains [11,50]. In our case, fine-tuning serves to adapt a general model into a specialized auditor capable of identifying vulnerabilities for smart contracts. Baseline LLMs used in a zero-shot configuration frequently misclassify safe constructs as unsafe or overlook non-obvious security flaws [18]. This stems from the model’s lack of exposure to Solidity’s nuanced semantics,

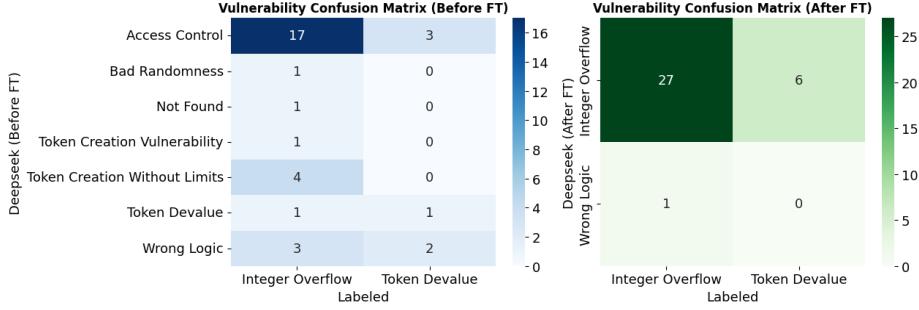


Fig. 1: Confusion matrices of vulnerability classification before and after fine-tuning (FT). The left matrix shows DeepSeek’s baseline predictions, where diverse categories such as “Access Control,” “Token Creation Vulnerability,” and “Wrong Logic” were often misclassified as “Integer Overflow” or “Token Devalue.” After fine-tuning (right), the model demonstrates substantially improved discrimination, with most cases of “Integer Overflow” correctly identified and reduced confusion across categories.

where superficially similar constructs can differ drastically in safety. For example, Kharkar et al. [27] report that an unadapted transformer mislabels many benign patterns as vulnerabilities due to spurious correlations, while a fine-tuned model more accurately separates true bugs from noise.

Figure 1 contrasts the a baseline LLM and our fine-tuned LLM on labeled smart contract data. The empirical results show that fine-tuning substantially aligns model predictions with true vulnerability types, reducing confusion between unrelated categories. For instance, the baseline model frequently misinterprets integer overflow vulnerabilities as benign logical issues or misclassifies unchecked call patterns. After fine-tuning, the Auditor correctly identifies these vulnerabilities with significantly fewer false positives, reflecting improved semantic discrimination.

3.2 Efficient Ensemble Learning

Fine-tuning an LLM on vulnerability-detection data improves domain specificity, yet relying on a single adapted model can leave systematic biases toward certain vulnerability types or datasets. Empirical studies report large run-to-run variability and low inter-model agreement for deep learning-based vulnerability detectors, and show that models struggle on under-represented CWEs and out-of-distribution (OOD) code [38,6,13].

Ensemble learning offers a practical mechanism to counter these limitations by aggregating predictions from multiple independently adapted models. The core idea is that heterogeneous base learners capture different error patterns. When combined, their complementary strengths can reduce variance and mitigate systematic, per-class biases, thereby improving overall robustness [12,49].

However, classical ensemble architectures can impose substantial computational burdens: training and serving many large-scale LLMs becomes infeasible under limited compute, making efficiency a key consideration in our ensemble design [16].

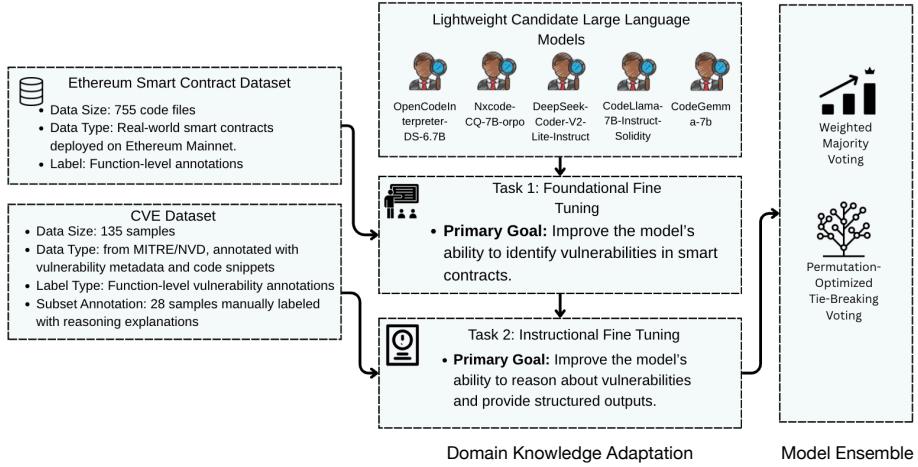


Fig. 2: Overview of LLMBugScanner, which consists of two stages: domain knowledge adaptation and model ensemble to improve the effectiveness of smart contract vulnerability detection.

4 Methodology

As presented in Figure 2, we introduce LLMBugScanner to address the motivation described earlier. LLMBugScanner comprises two synergistic components: (1) domain knowledge adaptation, (2) model ensemble.

Domain Knowledge Adaptation. To broaden the coverage of vulnerabilities to reduce bias, we adopt two real-world datasets of smart contracts. The details of two datasets are shown in Figure 3:

- **Ethereum Smart Contract Dataset:** 775 Solidity files labeled with vulnerabilities, used to improve general code interpretation.
- **CVE Solidity Subset:** 28 files (25% of full dataset), each annotated with a single vulnerability, used for instructional fine-tuning.

For each dataset, labels are transformed into structured prompts for the fine-tuning process. For the Ethereum Smart Contract dataset, contract names and

vulnerability lists are extracted. For the CVE dataset, vulnerability type, function name, and description are included. Variables are dynamically replaced with actual dataset values during fine-tuning. To improve computational efficiency, we employ LoRA [16], which injects low-rank trainable adapters into attention and feed-forward projection layers, enabling efficient fine-tuning with only a fraction of parameters modified [16].

For fine-tuning, the auditor prompt specifies (1) task instructions with a restricted set of vulnerability types, (2) the number of vulnerabilities to return, (3) a JSON output schema with error handling, and (4) the full smart contract code as input, shown as in Figure 4.

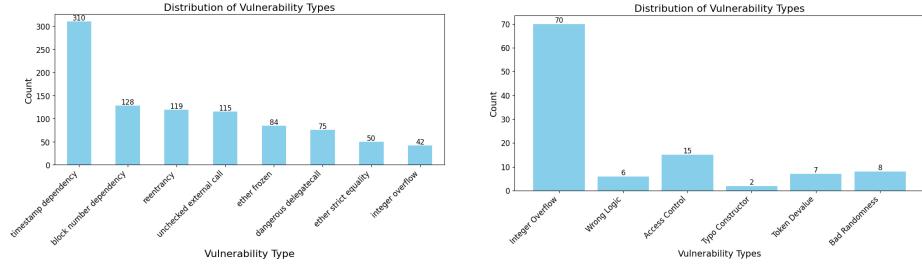


Fig. 3: Comparative distributions of vulnerability categories across the Ethereum and CVE datasets.

Model Ensemble. We select top-performing models from the OpenCodeLLM leaderboard for parameter-efficient fine-tuning:

- AlfredPros/CodeLlama-7b-Instruct-Solidity [3]
- m-a-p/OpenCodeInterpreter-DS-6.7B [47]
- NTQAI/Nxcode-CQ-7B-orpo [34]
- deepseek-ai/DeepSeek-Coder-V2-Lite-Instruct (15.7B) [2]
- google/codegemma-7b [10]

Individual LLM predictions are aggregated through multiple ensemble techniques. These strategies allow us to exploit complementary strengths of different models and to reduce noise from individual misclassifications.

Method 1: Weighted Majority Voting. We assign each model a weight based on its rank or performance (e.g., higher-performing models get higher weights). For each predicted (vulnerability, function) pair, the model’s vote is multiplied by its weight. These weighted votes are then summed across all models. The final

Task Instructions:

Requirement: You are a smart contract auditor. Identify {topk} most severe vulnerability in the provided smart contract. Ensure it is exploitable in real world and beneficial to attackers. Restrict your identification to these vulnerability types: Integer Overflow, Wrong Logic, Bad Randomness, Access Control, Typo Constructor, Token Devalue.

Output Format:

You should ONLY output in below JSON format:

```
{  
    "output_list": [  
        {  
            "function_name": "<func name>",  
            "vulnerability": "<short desc>",  
            "reason": "<reason>"  
        },  
        {  
            "function_name": "<func name>",  
            "vulnerability": "<short desc>",  
            "reason": "<reason>"  
        }  
    ]  
}
```

Note: If no vulnerabilities are found, output an empty JSON:

```
{"output_list": []}
```

Full Code Input:

```
{code}
```

Your Output:

Fig. 4: Auditor prompt

score for a pair j is:

$$\text{Score}_j = \sum_{i \in M} w_i \cdot v_{ij}$$

where M is the set of models, w_i is the weight assigned to model i , and $v_{ij} \in \{0, 1\}$ indicates whether model i voted for pair j . We then select the top five pairs with the highest total weighted scores as the ensemble predictions. This approach emphasizes high-quality models while still benefiting from consensus.

Method 2: Permutation-Optimized Tie-Breaking Voting. We also evaluate a permutation-optimized, tie-breaking voting strategy. Here, we first compute unweighted votes:

$$\text{Score}_j = \sum_{i \in M} v_{ij}$$

If multiple pairs tie in Score_j , the tie is broken using a learned model priority order $\pi(i)$ (lower is higher priority). This order is optimized on a validation set to maximize ensemble accuracy on a held-out subset. Ties in Score_j are broken by the earliest model (lowest $\pi(i)$) that voted for the pair. This mechanism preserves the simplicity of unweighted voting while leveraging model-specific knowledge to resolve ambiguous cases.

In practice, we apply both methods and compare results on a validation set to choose the optimal ensemble for final deployment. Weighted voting tends to improve precision by giving more influence to strong models, while permutation-optimized tie-breaking improves recall by resolving conflicts in favor of historically reliable models. We also experiment with different values of k (number of top pairs retained) to assess trade-offs between coverage and accuracy.

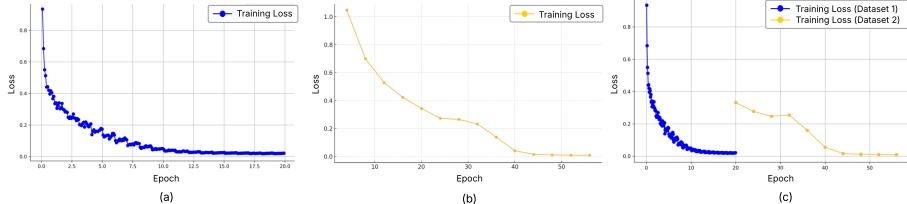


Fig. 5: Training loss curves for models fine-tuned on Ethereum only (a), CVE only (b), and both datasets (c). The plots show rapid loss reduction during the early epochs with continued convergence across all settings. Notably, when fine-tuned on both datasets, the loss for the second dataset starts lower than in the standalone training scenario, highlighting the effectiveness of pre-training on the first dataset.

5 Experiment

In this section, we validate the previous analysis and the efficacy of LLM-BugScanner via experimental results.

5.1 Experimental Settings

Dataset: We evaluate LLMBugScanner on a 80% random split of CVE-Solidity dataset [18], which contains 108 annotated smart contract files with each labeled with a single vulnerability type. The dataset captures diverse Ethereum smart contract vulnerabilities and serves as the benchmark for all model comparisons, shown in Fig. 3b.

Models and Fine-Tuning: Five open-source light-weight large language models (LLMs) were selected based on their performance on code understanding tasks: AlfredPros/CodeLlama-7b-Instruct-Solidity, m-a-p/OpenCodeInterpreter-DS-6.7B, NTQAI/Nxcode-CQ-7B-orpo, deepseek-ai/DeepSeek-Coder-V2-Lite-Instruct, and google/codegemma-7b. Each model was fine-tuned using LoRA [16] on two complementary datasets: the Ethereum smart contract dataset (general code interpretation) and a 25% subset of the CVE dataset (instructional fine-tuning). LoRA hyperparameters were chosen based on reported best-performing settings in prior work. All models were evaluated in bfloat16 precision.

As shown in Fig. 5, models first fine-tuned on the Ethereum dataset exhibited substantially lower initial loss and faster convergence when subsequently trained on the CVE subset, compared to models trained on either dataset independently. This indicates that knowledge gained from the domain-general code representation phase transfers effectively to security-specific instruction tuning, reinforcing the benefit of sequential fine-tuning for cross-dataset generalization.

Fine-Tuning Configuration. All models are fine-tuned sequentially (Ethereum Smart Contract dataset → CVE subset dataset) using LoRA. The optimal hyperparameters were determined through a series of experiments varying learning rates, LoRA ranks, and batch sizes, with the final configuration selected based on validation performance and training stability.

Auditor Prompt: For evaluation, we employ a code auditing prompt that specifies (1) the role of the model as a vulnerability auditor, (2) the exact number of vulnerabilities to return, and (3) a strict JSON output format with error handling. The full smart contract code is provided as input in Fig. 4.

Experiment Settings: To compare auditing strategies, we implemented multiple configurations:

- **Single-Model Baselines:** Each LLM independently identifies vulnerabilities using the auditor prompt.
- **Single-Model Finetuned:** Each fine-tuned LLM independently identifies vulnerabilities using the auditor prompt.
- **Ensemble-Model Weighted Voting:** Models are assigned weights based on their individual top- k hit rates, and the highest-scoring vulnerability predictions are selected.

Table 2: LoRA Fine-Tuning Configuration (NXcode as Example)

| Component | Value |
|-----------------------|-----------------------------|
| Base Model | NTQAI/Nxcode-CQ-7B-orpo |
| LoRA Rank (r) | 32 |
| LoRA Modules | Attention + MLP + LM Head |
| Precision | bfloat16 |
| Batch Size | 32 |
| Epochs | 40 |
| Learning Rate | 2e-4 |
| Gradient Accumulation | 2 |
| Trainer | <code>trl.SFTTrainer</code> |

Table 3: Parameter settings for generation

| Parameter | Value |
|----------------------|-------|
| max_new_tokens | 800 |
| temperature | 0.1 |
| top_k | 10 |
| top_p | 0.95 |
| num_return_sequences | 1 |
| repetition_penalty | 1.5 |

– **Ensemble-Model Permutation-optimized Tie-Breaking Voting:** Permutation-optimized model ordering is used to break ties after unweighted voting, improving ensemble performance.

Evaluation Metrics: We report Top- k hit rates under a strict match criterion (function name and vulnerability type) and under a soft cosine-similarity match criterion between model-generated descriptions and ground-truth labels at thresholds 0.5, 0.7, and 0.9. Both Top-1 and Top-5 hit rates are presented.

Implementation Details: All experiments were run on NVIDIA H100 GPUs. For all models, the generation parameters are set as shown in Table 3.

This setup enables direct comparison between single-model performance and ensemble strategies, quantifying the gains from fine-tuning and ensembling on smart contract vulnerability detection.

5.2 Performance Comparison

The results are presented in Table 4, where we can make several observations:

– **Fine-tuning boosts Top-hit performance.** Across almost all models and thresholds, fine-tuned versions achieve substantially higher Top-1 and Top-5 hit rates than their baselines. Gains are most pronounced at the moderate similarity threshold $t = 0.5$, showing that the fine-tuned models produce more relevant matches under relaxed and moderate criteria.

Table 4: Model Performance Comparison. For rows with cosine similarity (cs), a hit is counted when the description appears in the Top-1 or Top-5 and its similarity exceeds threshold $t \in \{0.5, 0.7, 0.9\}$. The last two rows show direct-match hit rates (no threshold). FT = finetuned. Overall, finetuned models (FT) consistently outperform their baselines across most thresholds, indicating improved retrieval precision after finetuning. Notably, ensemble methods achieve the highest Top-5 hit rates, suggesting complementary strengths among individual models.

| Metric | t | Code | | DeepSeek | | Gemma | | NXCode | | OpenInterpreter | | Perm-Opt | | Ensemble | | Weighted Ensemble | | |
|--------------------|-----|------|------|----------|------|-------|------|--------|------|-----------------|------|----------|------|----------|------|-------------------|----|--|
| | | Base | FT | Base | FT | Base | FT | Base | FT | Base | FT | Base | FT | Base | FT | Base | FT | |
| Top 1 hit (cs) | 0.5 | 0.20 | 0.41 | 0.38 | 0.56 | 0.15 | 0.37 | 0.26 | 0.47 | 0.20 | 0.59 | 0.28 | 0.33 | 0.28 | 0.25 | | | |
| Top 5 hit (cs) | 0.5 | 0.44 | 0.43 | 0.77 | 0.82 | 0.37 | 0.50 | 0.68 | 0.47 | 0.50 | 0.73 | 0.69 | 0.73 | 0.69 | 0.84 | | | |
| Top 1 hit (es) | 0.7 | 0.05 | | 0.29 | 0.12 | 0.22 | 0.02 | 0.29 | 0.09 | 0.34 | 0.04 | 0.45 | 0.06 | 0.16 | 0.05 | 0.07 | | |
| Top 5 hit (es) | 0.7 | 0.06 | | 0.32 | 0.20 | 0.33 | 0.05 | 0.37 | 0.31 | 0.34 | 0.18 | 0.49 | 0.32 | 0.49 | 0.31 | 0.34 | | |
| Top 1 hit (es) | 0.9 | 0.00 | | 0.08 | 0.00 | 0.00 | 0.00 | 0.21 | 0.00 | 0.06 | 0.00 | 0.24 | 0.00 | 0.05 | 0.00 | 0.00 | | |
| Top 5 hit (es) | 0.9 | 0.00 | | 0.09 | 0.00 | 0.00 | 0.00 | 0.21 | 0.00 | 0.06 | 0.00 | 0.25 | 0.00 | 0.20 | 0.00 | 0.00 | | |
| Top 1 hit (direct) | - | 0.06 | | 0.27 | 0.05 | 0.38 | 0.04 | 0.28 | 0.09 | 0.34 | 0.06 | 0.39 | 0.07 | 0.40 | 0.06 | 0.093 | | |
| Top 5 hit (direct) | - | 0.12 | | 0.31 | 0.14 | 0.56 | 0.11 | 0.38 | 0.41 | 0.34 | 0.27 | 0.41 | 0.44 | 0.60 | 0.44 | 0.56 | | |

- **Permutation-optimized vs. weighted ensemble.** The permutation-optimized ensemble attains consistently higher Top-1 rates than individual models at $t = 0.5\text{--}0.7$, whereas the weighted ensemble shows a clear advantage in Top-5 metrics (e.g., 0.84 at $t = 0.5$ vs. 0.73 for perm-opt). This suggests the perm-opt method prioritises precision on the single best candidate, while the weighted scheme increases recall across several candidates, boosting Top-5 performance.
- **Description similarity thresholds magnify differences.** As the cosine-similarity threshold increases from 0.5 to 0.9, hit rates collapse for all models—especially for Top-1. However, fine-tuned models and both ensemble variants retain a relative advantage even at $t = 0.7\text{--}0.9$, implying their descriptions are not only more likely to appear in the ranked list but also more semantically aligned with the ground truth.
- **Direct match versus similarity-based match comparison.** The ensemble model steadily improves direct match (by function and vulnerability name) in both top-1 and top-5 accuracy. However, in the description similarity match, it sometimes performs worse than the top-performing single model, in this case, *open-interpreter*. This indicates that there may be a misalignment between the models’ textual reasoning and their actual vulnerability/function identification, suggesting that some ensemble mechanisms amplify shallow consensus rather than deeper semantic understanding. Moreover, The ensemble could be overfitting to lexical overlap in predictions, leading to improved direct matching but weaker semantic alignment. Last, Incorporating semantic-aware aggregation (e.g., embedding-based or explanation-consistency weighting) might help align reasoning quality with identification accuracy in future iterations.

To summarize, these results suggest that fine-tuning and ensembling provide complementary advantages. Fine-tuning improves each model in detecting frequently occurring vulnerabilities and produces more semantically consistent

outputs, while the ensemble, especially the weighted version, broadens coverage across multiple plausible predictions, which is valuable for retrieval tasks where several closely matching answers can all be acceptable.

5.3 Case Study: best single model vs. best ensembled model

To move beyond aggregate metrics, we performed a case analysis comparing the best-performing single model with the ensemble model across four correctness scenarios: (1) both models incorrect, (2) best model correct but ensemble incorrect, (3) ensemble correct but best model incorrect, and (4) both models correct. This analysis does not simply describe the distributions of vulnerability types and functions, but reveals the mechanisms by which the ensemble improves or degrades detection performance.

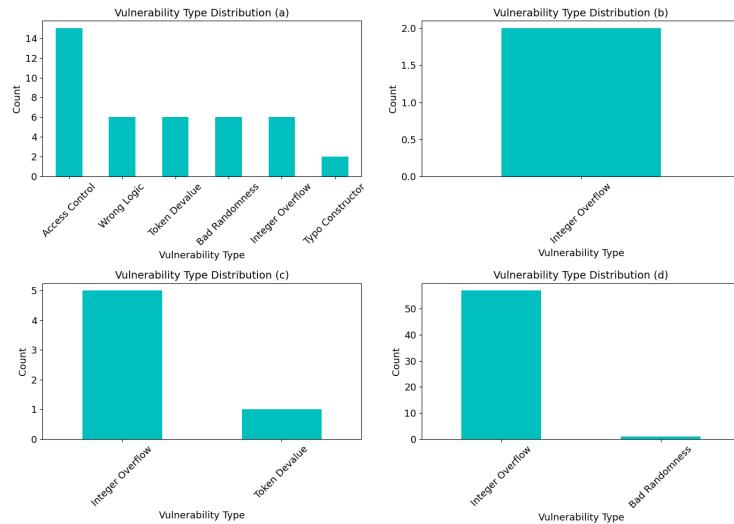


Fig. 6: Vulnerability-type distributions under different model agreement scenarios. (a) Cases where both the ensemble and DeepSeek are incorrect show a wider spread across vulnerability types, with Access Control dominating the errors. (b) Cases where DeepSeek is correct but the ensemble is incorrect are rare and concentrated on a single vulnerability type (Integer Overflow). (c) Cases where DeepSeek is incorrect but the ensemble is correct indicate that ensemble predictions recover some errors from DeepSeek, mainly on Integer Overflow and Token Devalue vulnerabilities. (d) Cases where both the ensemble and DeepSeek are correct are dominated by Integer Overflow, with a small fraction of Bad Randomness vulnerabilities, showing the strongest area of agreement.

Recovering High-Impact Vulnerabilities. A key insight from Figure 6 is that the ensemble recovers some vulnerabilities, particularly Integer Overflow and Token Devalue, that the best single model misses. However, there are two Integer Overflow cases missed where the best single model correct. The fact that Integer

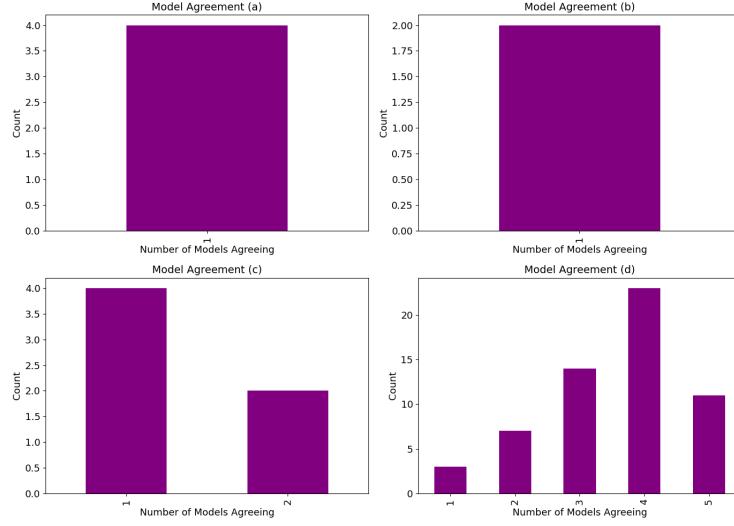


Fig. 7: Model agreement distribution under different model agreement scenarios. (a) Both the ensemble and DeepSeek are incorrect. (b) DeepSeek is correct but the ensemble is incorrect. (c) DeepSeek is incorrect but the ensemble is correct. (d) Both ensemble and DeepSeek are correct, showing consistently high agreement (4–5 models), demonstrating that strong consensus within the ensemble correlates with accurate predictions.

Overflow is the dominant case could lead to overfitting for all the models. This could explain the increase in correctly identified Integer Overflow cases by the ensemble model. When combined with the two Integer Overflow cases miss and two Token Devalue cases hit for ensemble model, ensemble model shows an ability to provide a less biased prediction while could lead to some mismatch and introduce some noise. On the other hand, both models have a hard time predicting other minority classes include Wrong Logic, Access Control, and Typo Construction. That shows that the ensemble model still cannot handle minority cases when the training set is limited and biased due to each model in the ensemble model faces same problem. One important vulnerabilities is Access Control. With a relative large sample number compared to Token Devalue and Bad Randomness, the models had a hard time to make a correct prediction in this specific type of vulnerabilities. Since each model are pre-trained with larger dataset, the problem could be the knowledge from the pre-training dataset are not enough for model to make correct decision. It could also imply that the Access Control cases are more complicated, and hard to be caught by current model we have.

Voting Distribution and Consensus Dynamics. The model agreement plots (Figure 7) quantify voting dynamics. When both models are wrong, agreement within the ensemble is minimal, indicating that diversity alone does not guarantee success on truly hard cases. Especially for minority case, the nature of lack of

training data make all models have a hard time to make good prediction. In this case, the ensemble itself cannot improve the performance of model prediction. When the best model is correct but the ensemble is wrong, agreement is also low, reflecting the dilution of a correct minority vote. By contrast, when the ensemble is correct and the single model is wrong, moderate agreement emerges; here, the ensemble is successfully amplifying weak but consistent votes across models. When both models are correct, consensus is highest, reflecting that easy or obvious cases drive strong agreement.

Overall, these observations provide two actionable insights for improving ensemble-based vulnerability detection. First, ensembles deliver the largest gains on high-severity, heterogeneous vulnerability types and functions, suggesting that model diversity is most valuable where classification requires reasoning across multiple semantic cues. Second, ensembles are vulnerable to overcorrection in low-diversity settings, implying that weighting or confidence calibration of constituent models could further improve performance.

6 Future Directions

Learning-Based Ensembles. While our current architecture uses rule-based ensembles over multiple fine-tuned LLMs, a more advanced path is to adopt a learning-based ensemble strategy. Instead of fixed rules to decide which model’s output to trust, one could train a *meta-classifier* that, given features of the input, predicts which LLM is most likely to produce a correct vulnerability diagnosis. This approach essentially turns the ensemble step into a learned decision problem. To do this reliably, one would require a larger labelled dataset: the data would have to be partitioned not just into train/validation/test, but also into a classifier-training split (for learning which model to trust) and a held-out final test split to evaluate end-to-end performance. The overhead is nontrivial, but this approach has been successful in other ensemble settings (e.g., stacking in machine learning) [36,44] and could adapt dynamically to different types of smart contract code.

Hallucination Mitigation. A second direction is to tackle hallucination and inconsistent outputs. In our experiments, we observe a $\sim 10\%$ hallucination rate across models—instances where the model invents vulnerabilities or outputs contradictory diagnoses. This likely stems from using lightweight fine-tuning with constrained adaptation capacity. Future work could mitigate hallucinations via several strategies: (i) *contrastive regularization*, where known safe code is paired explicitly against vulnerable code to penalize stray predictions [23]; (ii) *self-consistency verification*, where multiple prompting or sampling is used and only consensus outputs are accepted [42,5]; and (iii) integrating *symbolic consistency checks* as a filter layer—e.g., verifying that a flagged vulnerability is consistent with the control-flow or state transitions of the contract code. Combining LLM predictions with symbolic validators may reduce hallucination while retaining

flexibility. Further, self-reflective frameworks such as SaySelf [45] could be explored to help models estimate their own confidence and suppress low-certainty predictions.

Code Normalization and Canonicalization. A third promising direction is to enhance model robustness through code normalization and canonicalization. Variation in variable names, formatting, ordering of statements, or syntactic sugar can make it harder for an LLM to generalize vulnerabilities across code that is semantically equivalent but syntactically different. Introducing a normalization preprocessing step—such as renaming variables to canonical tokens, rewriting expressions to a normalized intermediate form, or applying AST reordering—could reduce the noise the model sees. Prior work on code normalization and embedding learning has shown improved generalization for downstream tasks [24,46,1]. One could even train a denoising autoencoder (or fine-tuned model) that maps arbitrary Solidity code into a canonical form before vulnerability detection. Doing so may boost the models’ robustness to superficial syntactic variation and improve generalization to unseen contract styles.

In summary, advancing our system along these paths—meta-learned ensembles, hallucination filtering, and normalization pipelines—offers a roadmap to more accurate, credible, and robust smart contract vulnerability detection.

7 Conclusion

We have presented LLMBugScanner, a systematic LLM-powered approach for smart contract vulnerability detection. The design of LLMBugScanner has several unique features. First, we effectively utilize domain knowledge adaptation through multi-stage fine-tuning across different benchmark datasets with parameter-efficient techniques. Second, we integrate domain-knowledge adaption with LLM-ensemble to enable LLMBugScanner to acquire complementary expertise across multiple independently pre-trained and fine-tuned LLMs over multiple general programming and security-specific datasets. Our LLM-ensemble method resolves reasoning conflicts among multiple LLMs through consensus convergence. Experimental results across multiple LLM families, including CodeLlama, DeepSeek, NXCode, and CodeGemma, show that LLMBugScanner offers consistent improvements in precision, recall, and robustness over single-model baselines studied in this paper, highlighting LLMBugScanner as a cost-effective, extensible, and trustworthy approach for LLM-based smart contract auditing.

Acknowledgements

This research is partially sponsored by the NSF CISE grants 2302720 and 2312758, an IBM faculty award, and a CISCO research grant in Edge AI.

References

1. Ahmad, W.U., Chakraborty, S., Ray, B., Chang, K.W.: Code normalization for machine learning models of code. In: Proceedings of the 45th International Conference on Software Engineering (ICSE). pp. 1–13 (2023)
2. AI, D.: Deepseek-coder-v2: Breaking the barrier of closed-source models. arXiv preprint arXiv:2406.11931 (2024), mixture-of-Experts code model family; supports e.g. textttDeepSeek-Coder-V2-Lite-Instruct
3. AlfredPros: Codellama-7b instruct solidity (fine-tuned code llama for solidity smart contracts). <https://huggingface.co/AlfredPros/CodeLLaMa-7b-Instruct-Solidity> (2024), fine-tuned 7B parameter model for Solidity, dataset 6,003 instruction/source-code pairs
4. Atzei, N., Bartoletti, M., Cimoli, T.: A survey of attacks on ethereum smart contracts (sok). In: Principles of Security and Trust, pp. 164–186. Springer (2017). doi:10.1007/978-3-662-54455-6_8
5. Chen, X., Aksitov, R., Alon, U., Ren, J., Xiao, K., Yin, P., Prakash, S., Sutton, C., Wang, X., Zhou, D.: Universal self-consistency for large language model generation. arXiv preprint arXiv:2311.17311 (2023), <https://arxiv.org/abs/2311.17311>
6. Chen, Y., Ding, Z., Alowain, L., Chen, X., Wagner, D.: 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. pp. 654–668 (2023)
7. ConsenSys Diligence: Mythril: Security analysis tool for ethereum smart contracts. <https://github.com/ConsenSys/mythril> (2018), accessed: 2025-10-06
8. vulnerability database, N.: Common vulnerabilities and exposures (cves), <https://cve.mitre.org/index.html>
9. David, I., Zhou, L., Qin, K., Song, D., Cavallaro, L., Gervais, A.: Do you still need a manual smart contract audit? arXiv preprint arXiv:2306.12338 (2023), <https://arxiv.org/abs/2306.12338>
10. DeepMind, G., AI, G.: Codegemma: Open code models based on gemma. arXiv preprint arXiv:2406.11409 (2024), code-specialised variant of the Gemma model family
11. Devlin, J., Chang, M.W., Lee, K., Toutanova, K.: Bert: Pre-training of deep bidirectional transformers for language understanding. NAACL-HLT pp. 4171–4186 (2019), <https://aclanthology.org/N19-1423>
12. Dietterich, T.G.: Ensemble methods in machine learning. In: International workshop on multiple classifier systems. pp. 1–15. Springer (2000)
13. Du, X., Wen, M., Zhu, J., Xie, Z., Ji, B., Liu, H., Shi, X., Jin, H.: Generalization-enhanced code vulnerability detection via multi-task instruction fine-tuning. arXiv preprint arXiv:2406.03718 (2024)
14. Durieux, T., Ferreira, J.F., Abreu, R., Cruz, P.: Empirical review of automated analysis tools on 47,587 ethereum smart contracts. In: Proceedings of the 42nd International Conference on Software Engineering. pp. 530–541. ACM (2020). doi:10.1145/3377811.3380364
15. Feist, J., Grieco, G., Groce, A.: Slither: a static analysis framework for smart contracts. In: 2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB). pp. 8–15. IEEE (2019)
16. Hu, E.J., Shen, Y., Wallis, P., Allen-Zhu, Z., Li, Y., Wang, S., Wang, L., Chen, W.: Lora: Low-rank adaptation of large language models. arXiv preprint arXiv:2106.09685 (2021), <https://arxiv.org/abs/2106.09685>

17. Hu, S., Huang, T., Chow, K.H., Wei, W., Wu, Y., Liu, L.: Zipzap: Efficient training of language models for large-scale fraud detection on blockchain. In: Proceedings of the ACM Web Conference 2024. pp. 2807–2816 (2024)
18. Hu, S., Huang, T., İlhan, F., Tekin, S.F., Liu, L.: 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). pp. 297–306. IEEE (2023)
19. Hu, S., Huang, T., Liu, G., Kompella, R.R., İlhan, F., Tekin, S.F., Xu, Y., Yahin, Z., Liu, L.: A survey on large language model-based game agents. arXiv preprint arXiv:2404.02039 (2024)
20. Hu, S., Zhang, Z., Luo, B., Lu, S., He, B., Liu, L.: Bert4eth: A pre-trained transformer for ethereum fraud detection. In: Proceedings of the ACM Web Conference 2023. pp. 2189–2197 (2023)
21. Iuliano, A., Bartoletti, M., Bracciali, A., Marchesi, M., Ibba, S., Lunesu, I., Pinna, A.: A comprehensive survey of smart contract vulnerabilities and detection tools. ACM Computing Surveys **56**(4), 1–42 (2024). doi:10.1145/3631785, <https://dl.acm.org/doi/10.1145/3631785>
22. Iuliano, G., Di Nucci, D.: Smart contract vulnerabilities, tools, and benchmarks: An updated systematic literature review. arXiv preprint **abs/2412.01719** (2024), <https://arxiv.org/abs/2412.01719>
23. Ji, C., Yang, S., Sun, H., Zhang, Y.: Applying contrastive learning to code vulnerability type classification. In: Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing. pp. 11942–11952. Association for Computational Linguistics (Nov 2024)
24. Kanade, A., Maniatis, P., Balachandran, V., Shi, Z., Chou, S.: Learning and evaluating contextual embedding of source code. In: ICML (2020)
25. Karanjai, R., Blackshear, S., Xu, L., Shi, W.: Smartify: A multi-agent framework for automated vulnerability detection and repair in solidity and move smart contracts. arXiv preprint arXiv:2502.18515 (2025)
26. Khan, S.N., Loukil, F., Ghedira-Guegan, C., Benkhelifa, E., Bani-Hani, A.: Blockchain smart contracts: Applications, challenges, and future trends. Peer-to-peer Networking and Applications **14**(5), 2901–2925 (2021)
27. Kharkar, A., Rawat, A., Sharma, V., Foster, J.S., Ray, B.: Learning to reduce false positives in static analysis. In: Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering (ASE 2022). pp. 1291–1302 (2022). doi:10.1145/3551349.3556951
28. Li, X., Wang, S., Zeng, S., Yang, Y.: A survey on llm-based multi-agent systems: Workflow, infrastructure, and challenges. IEEE Transactions on Artificial Intelligence (2024), early Access
29. Li, Z., Zou, D., Xu, S., Ou, X., Wang, S., Deng, Z., Zhong, Y., Jin, H.: Vuldeepecker: A deep learning-based system for vulnerability detection. In: NDSS Symposium (2018), https://ndss-symposium.org/wp-content/uploads/2018/02/ndss2018_03A-2_Li_paper.pdf
30. Luu, L., Chu, D.H., Olickel, H., Saxena, P., Hobor, A.: Making smart contracts smarter. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security. pp. 254–269 (2016)
31. Mehar, M., Ostrovski, I., Škenda, M., Røstad, L., Cavalli, A., Lodder, A., Parisi, F., Jean, F.: The dao: A decentralized autonomous organization — a case study of the smart contract use in the blockchain. SSRN Electronic Journal (2017). doi:10.2139/ssrn.3017682

32. Mossberg, M., DeCarlo, J., Muench, M., Fitzpatrick, B., Redini, N., Giuffrida, C., Cavallaro, L.: Manticore: A user-friendly symbolic execution framework for binaries and smart contracts. In: Proceedings of the 34th Annual Computer Security Applications Conference (ACSAC). pp. 590–602 (2018)
33. Nikolić, I., Kolluri, A., Sergey, I., Saxena, P., Hobor, A.: Finding the greedy, prodigal, and suicidal contracts at scale. In: Proceedings of the 34th Annual Computer Security Applications Conference. pp. 653–663. ACM (2018). doi:10.1145/3274694.3274743
34. NTQAI: Nxcode-cq-7b-orpo (monolithic preference-optimization fine-tune of codeqwen1.5-7b on 100k high-quality ranking samples). <https://huggingface.co/NTQAI/Nxcode-CQ-7B-orpo> (2025), fine-tune without reference model; ranking data set 100k
35. Rameder, H., Di Angelo, M., Salzer, G.: Review of automated vulnerability analysis of smart contracts on ethereum. *Frontiers in Blockchain* **5**, 814977 (2022). doi:10.3389/fbloc.2022.814977
36. Ridoy, S.Z., Shaon, M.S.H., Cuzzocrea, A., Akter, S.: Enstack: An ensemble stacking framework of large language models for enhanced vulnerability detection in source code. arXiv preprint arXiv:2411.16561 (2024), <https://arxiv.org/abs/2411.16561>
37. Roziere, B., Gehring, J., Gloeckle, F., Sootla, S., Gat, I., Tan, X.E., Adi, Y., Liu, J., Sauvestre, R., Remez, T., et al.: Code llama: Open foundation models for code. arXiv preprint arXiv:2308.12950 (2023)
38. Steenhoek, B., Rahman, M.M., Jiles, R., Le, W.: An empirical study of deep learning models for vulnerability detection. In: 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE). pp. 2237–2248. IEEE (2023)
39. Sun, Y., Wu, D., Xue, Y., Liu, H., Wang, H., Xu, Z., Xie, X., Liu, Y.: When gpt meets program analysis: Towards intelligent detection of smart contract logic vulnerabilities in gptscan. arXiv preprint arXiv:2308.03314 (2023)
40. Tsankov, P., Dan, A.M., Drachsler-Cohen, D., Gervais, A., Buenzli, F., Vechev, M.T.: Securify: Practical security analysis of smart contracts. In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security. pp. 67–82. ACM (2018). doi:10.1145/3243734.3243780
41. Vidal, A., Bartoletti, M., Pinna, A.: Vulnerability detection in smart contracts with large language models. arXiv preprint arXiv:2403.08124 (2024), <https://arxiv.org/abs/2403.08124>
42. Wang, X., Wei, J., Schuurmans, D., Le, Q.V., Chi, E.H., Narang, S., Chowdhery, A., Zhou, D.: Self-consistency improves chain of thought reasoning in language models. arXiv preprint arXiv:2203.11171 (2022), <https://arxiv.org/abs/2203.11171>
43. Wei, Z., Sun, J., Zhang, Z., Zhang, X.: Llm-smartaudit: Advanced smart contract vulnerability detection via multi-agent collaboration. In: Proceedings of the 2024 IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER) (2024)
44. Wolpert, D.H.: Stacked generalization. *Neural Networks* **5**(2), 241–259 (1992)
45. Xu, T., Wu, S., Diao, S., Liu, X., Wang, X., Chen, Y., Gao, J.: Saysself: Teaching llms to express confidence with self-reflective rationales. arXiv preprint arXiv:2405.20974 (2024), <https://arxiv.org/abs/2405.20974>
46. Zhang, Y., Hu, X., Zhao, S., Chen, H., Wang, Y., Liu, S., Zhou, D., Zhou, M.: Codet5+: Open code large language models for code understanding and generation. arXiv preprint arXiv:2305.07922 (2023), <https://arxiv.org/abs/2305.07922>

47. Zheng, T., et al.: Integrating code generation with execution and refinement. In: Findings of ACL. pp. ???–???. (2024), used for model series textttm-a-p/OpenCodeInterpreter-DS-6.7B
48. Zheng, Z., Xie, S., Dai, H.N., Chen, X., Wang, H.: An overview of blockchain and smart contract security: Challenges and opportunities. IEEE Access **7**, 45441–45471 (2020)
49. Zhou, Z.H.: Ensemble methods: foundations and algorithms. CRC press (2025)
50. Zhuang, Y., Liu, Q., Hu, X., Yin, J.: A comprehensive survey of smart contract security: Vulnerabilities, attacks, and tools. IEEE Access **8**, 219582–219601 (2020). doi:10.1109/ACCESS.2020.3042440
51. Zou, D., Wang, S., Xu, S., Li, Z., Jin, H.: μ vuldeepecker: A deep learning-based system for multiclass vulnerability detection. IEEE Transactions on Dependable and Secure Computing **18**(5), 2224–2236 (2021). doi:10.1109/TDSC.2020.2973304