DefixGuard: LLM-Driven Smart Contract Auditing and Security Assurance

Sainesh Nakra[1] and Dr. Almudena Konrad[2]

Northeastern University, Boston, MA, USA
nakra.s@northeastern.edu, a.konrad@northeastern.edu

Abstract. Smart contracts form the backbone of Decentralized Finance (DeFi) platforms, but their complexity makes them highly vulnerable to exploits. Current automated tools for smart contract auditing often fail to detect complex vulnerabilities and generate a high number of false positives. To address these challenges, we present DefixGuard, a scalable and automated framework for smart contract auditing that leverages the strengths of Large Language Models (LLMs) and static analysis tools. By integrating a multi-LLM architecture and a weighted consensus algorithm, DefixGuard significantly enhances the accuracy of vulnerability detection while automating the auditing process. Our framework goes beyond traditional auditing by generating detailed, multilayered audit reports. These reports not only identify vulnerabilities but also include actionable remediation steps, offering comprehensive security assessments, risk evaluations, and clear recommendations for not only developers to address their issues but also for everyday DeFi users to understand whether they should invest in a project or not. Evaluation of DefixGuard on real-world DeFi smart contracts shows a precision of 94%, outperforming existing tools like Slither and Mythril. Despite its strengths, challenges remain in addressing certain attack vectors such as oracle manipulations. Future work will focus on refining detection models and improving coverage, making DefixGuard a promising solution for large-scale, automated smart contract auditing in the DeFi ecosystem.

1 Introduction

Decentralized Finance (DeFi) represents a transformative shift in the financial industry, using blockchain technology to create an open, transparent, and borderless financial system. Unlike traditional financial systems, which are controlled by centralized entities such as banks and governments to handle transactions and maintain trust, DeFi operates on a decentralized network where participants interact directly with one another [1]. This decentralization of the financial system democratizes access to financial services, makes transaction fees transparent and increases the efficiency of financial transaction by eliminating intermediaries [2].

At the core of DeFi are smart contracts—self-executing agreements encoded on blockchain platforms. The concept of smart contracts, first introduced by

Nick Szabo in 1997, refers to digital agreements that automatically enforce predefined rules and conditions without the need for intermediaries [3]. These contracts bring transparency by automating transactions, but the immutability of smart contracts—once deployed, their code cannot be altered—along with it's complexity to an average investor introduces significant security challenges. Any embedded vulnerabilities or errors in the smart contract code are irreversible and can be exploited by malicious actors, leading to substantial financial losses [5]. A prime example is the WazirX hack, which resulted in a loss of $230 million, calling to attention the urgency and importance of securing smart contracts and analyzing their behavior before interacting with them in the DeFi ecosystem.

The rise of DeFi has also led to an increase in attacks targeting these platforms, as illustrated by a recent Systematization of Knowledge (SoK) study that categorized various DeFi attacks and their implications [6]. In 2022, DeFi platforms reported losses exceeding $3.8 billion due to smart contract exploits, marking a 47% increase from the previous year [7]. These incidents emphasize the urgent need for robust security measures to protect users and maintain trust within the DeFi space.

Research on learning meaningful code changes via neural machine translation has shown that machine learning models can effectively understand and generate code by learning from patterns in existing codebases [15]. This capability is crucial for smart contract auditing, where subtle code changes can distinguish between identifying vulnerabilities and missing them entirely. The approach developed by Tufano et al. [15] shows that when properly trained, machine learning models can capture the context and intent behind code changes, making them valuable tools for enhancing the security and reliability of smart contracts.

In response to these security challenges and the research done with neural networks in this domain, the research community has increasingly turned to advanced machine learning techniques. Large Language Models (LLMs), trained on vast datasets, have shown promise in analyzing and understanding complex code structures within smart contracts [8].

Despite these advancements, there is still no automated tool for smart contract auditing that effectively detects vulnerabilities across the diverse range of contracts deployed in DeFi and that can generate reports that can be understood by everyday DeFi users. To address this gap, we propose the DefixGuard framework, a novel approach that uses a multiple LLM architecture to enhance smart contract security. Our framework integrates traditional auditing tools with advanced machine learning models, providing a comprehensive, automated solution for auditing smart contracts, thus improving the security and reliability of DeFi platforms. The contributions of this paper are as follows:

Design and Implementation: This paper presents *DefixGuard*, a novel framework that enhances the process of smart contract auditing by integrating state-of-the-art Large Language Models (LLMs), such as GPT-4 [23], Gemini [22], and LLaMA 3.1b [21], with static analysis tools like Slither [26] and Mythril [27]. DefixGuard leverages the complementary strengths of LLMs and static tools, where both sets of tools analyze smart contracts independently. The

framework subsequently combines these outputs through a custom algorithm, to detect complex vulnerabilities, such as missing sanity checks and malicious logic-that cannot be done using static tools—also reduces false positives—a prevalent issue in automated smart contract auditing using LLMs. DefixGuard offers an extendable framework with improved accuracy and reliability in vulnerability detection.

Scalability and Automation: DefixGuard is designed with scalability at its core, enabling the efficient auditing of multiple smart contracts simultaneously. The framework automates the auditing process, removing the need for manual intervention and makes consistent performance possible even as the number of contracts increases. The auditing process is structured in a pipeline format, which optimizes resource allocation and the makes the system scalable.

Multi-LLM Architecture with Weighted Majority Consensus Algorithm: Our multi-LLM architecture introduces an adaptation of weighted majority consensus algorithm [9], assigning varying weights to Slither, Mythril, GPT-4, and LLaMA based on their reliability. Vulnerabilities are flagged only when the weighted score exceeds the 51% threshold-in accordance with the weighted majority consensus algorithm [9], improving detection accuracy and minimizing false positives.

Synergistic Use of Static Tools and LLMs: By combining static tools, which excel at detecting well-understood vulnerabilities, with LLMs that specialize in complex logic and business rule detection, DefixGuard ensures comprehensive coverage. This hybrid approach addresses a broader range of vulnerabilities that each method might miss individually.

Audit-Style Report Generation: The reports generated by DefixGuard provide rich, multi-layered analysis, offering detailed security evaluations, risk assessments, and actionable remediation steps. This combination of static and LLM-driven insights delivers more comprehensive and actionable security reports than traditional auditing tools.

2 Related Work

The field of smart contract auditing has advanced considerably, particularly with the adoption of machine learning techniques. Despite this progress, there remains a critical need for a reliable, automated, systematic, and comprehensive approach capable of accurately identifying vulnerabilities and producing industry-standard audit reports. Ivan Popchev and his team proposed a structured five-step audit plan, applied to two smart contracts, which laid the foundation for smart contract auditing [16]. However, this approach relies heavily on manual audits, which are not feasible for large-scale applications due to scalability concerns and the potential for human error.

Our research builds on the efforts of utilizing Machine Learning for smart contract auditing by utilizing Large Language Models (LLMs), which can analyze the context and intricacies of smart contract code. LLMs have demonstrated effectiveness in detecting vulnerabilities, as shown in studies focused on

LLM-powered smart contract vulnerability detection [10]. However, LLMs can generate high false positive rates, a problem that can be mitigated by integrating security analysis tools like Slither and Mythril within the LLM framework [28].

Another challenge in smart contract auditing is the need to identify multiple types of vulnerabilities across different contracts. A 2022 study introduced a multi-task learning approach, where multiple tasks are learned concurrently to improve the model's generalization capabilities [29]. Our research extends this approach by incorporating multi-task learning into our framework, ensuring that LLMs can identify a wide range of vulnerabilities, including behavioral issues and potential malicious intent.

In addition to detecting vulnerabilities, providing detailed explanations for their existence is essential in smart contract auditing. A study from March 2024 explored the role of fine-tuned LLMs and agents in auditing smart contracts, emphasizing the importance of offering justifications for detected vulnerabilities [30]. Building on this approach, *DefixGuard* enhances both vulnerability detection and explanation by leveraging a combination of advanced LLMs and static analysis tools. By integrating outputs from multiple audit tools, such as Slither and Mythril, and synthesizing this data with LLM-driven insights, *DefixGuard* not only identifies vulnerabilities but also generates comprehensive audit reports, ensuring that developers receive thorough justifications and a holistic analysis of the smart contract's security posture.

Despite the potential of LLMs to identify vulnerabilities, their tendency to produce high false positive rates makes it necessary to include an additional manual audit step in automated workflows. A recent study in 2023 evaluated the performance of LLMs such as GPT-4 and Claude as smart contract auditors, highlighting the need for manual intervention [24]. The study utilized a database of 52 vulnerable smart contracts. While our research primarily focuses on the implementation of a multimodal, scalable, automated framework—*DefixGuard*—and its ability to generate detailed vulnerability reports efficiently, we are also leveraging a subset of the same database along with a few real world widely accepted solidity projects to audit and validate the framework's performance.

3 Datasets

In this section, we present multiple smart contract vulnerabilities relevant to the automated auditing framework *DefixGuard* along with the complete database. These vulnerabilities were chosen for their frequency and financial impact on Ethereum-based smart contracts [24].

Our dataset consists of 27 real-time smart contract projects from Ethereum, carefully selected to represent a wide array of real-world vulnerabilities. This project can be directly extended to any blockchain that is EVM-compatible, working for all blockchains that utilize the Ethereum Virtual Machine (EVM), which is widely adopted for its secure and decentralized execution of smart contracts. This dataset is derived from a study that includes instances of 52 exploited smart contract projects [24]. These contracts provide the necessary complexity

and diversity to rigorously evaluate the effectiveness of *DefixGuard's* multimodal auditing approach.

The framework is exposed to a realistic threat environment by being validated against a subset that mirrors actual attack vectors seen in DeFi and other blockchain use cases. This helps evaluate it objectively in detecting not only common vulnerabilities like reentrancy but also detecting edge cases where the smart contract logic can lead to locked or frozen funds.

The following vulnerabilities represent the major issues identified in the database:

Reentrancy Attacks: A reentrancy attack occurs when an attacker exploits a contract by repeatedly invoking a function before previous executions complete. This typically happens when external calls are made before the contract updates its internal state. By exploiting this timing, the attacker can drain funds. For example, the infamous Decentralized Autonomous Organization (DAO) hack, attackers exploited a reentrancy vulnerability to repeatedly call the withdrawal function, draining the contract's funds [31]. In the case of a reentrancy attack, the vulnerability allows attackers to repeatedly call critical functions, draining the contract's funds before the contract can update its state.

On-chain Oracle Manipulation: Oracle manipulation happens when attackers tamper with or falsify data from oracles, which feed external data (such as prices) into the contract. By providing incorrect data, attackers can manipulate contract logic, especially in DeFi applications. For example, in the bZx attack, the attacker manipulated price data from an oracle, allowing them to take out a disproportionately large loan by misleading the contract about collateral values [32]. In the context of oracle manipulation, the attacker distorts the data fed into the contract, leading to decisions based on faulty information that can have dire financial implications.

Governance Attacks: Governance attacks occur when an entity accumulates enough governance tokens to manipulate decision-making processes in a decentralized protocol, enabling them to push malicious proposals or actions. For example, a malicious actor could amass governance tokens to push a proposal that diverts funds from a treasury or alters contract logic in their favor. In the scenario of a governance attack, the vulnerability arises from the attacker's ability to influence protocol governance, allowing them to pass unauthorized proposals that benefit themselves at the expense of the community.

Token Standard Incompatibility: Incompatibility with established token standards (e.g., ERC-20 or ERC-721) can lead to issues like failed transfers or broken token logic. These vulnerabilities happen when smart contracts incorrectly handle token behaviors, such as transfer mechanisms. For example, a non-standard ERC-20 token could fail to handle decimals correctly, resulting in incorrect balances or failed transfers [34]. In instances of token standard incompatibility, the failure to adhere to expected token standards causes unexpected behaviors, such as incorrect token transfers or lost tokens during transactions.

Absence of Logic/Sanity Checks: Contracts that lack essential logic checks allow attackers to manipulate functions in unintended ways, leading to actions like unauthorized withdrawals or contract manipulation. For example, a contract without a proper withdrawal limit check could allow users to withdraw more funds than they have deposited, resulting in significant financial losses. In cases of absence of logic/sanity checks, the lack of proper validations enables attackers to exploit the contract's functionality, allowing actions outside intended parameters, such as excessive withdrawals.

Delegatecall Injection: This vulnerability arises when a contract unsafely uses the `delegatecall` function, allowing external malicious code to execute within the context of the calling contract. This can manipulate the contract's state and assets. For example, in the Parity Wallet Hack, a delegatecall injection allowed attackers to take control of the wallet's contract and drain funds [35]. In the event of delegatecall injection, the contract's unsafe use of `delegatecall` provides a vector for attackers to execute malicious code, compromising the integrity and state of the original contract.

In future iterations, we will expand the dataset to include more exploited contracts projects from public repositories, further testing *DefixGuard's* ability to audit large-scale systems efficiently. Our scaling efforts will also focus on evaluating the framework's performance with multiple contracts analyzed simultaneously, ensuring it remains scalable and effective in a production environment.

4 Methodology

Our approach, illustrated in Figure 1, combines static and dynamic analysis tools with advanced Large Language Models (LLMs) to perform comprehensive audits of smart contracts. Unlike conventional methods that merely classify contracts as safe or vulnerable, our framework provides detailed audit reports. These reports cover vulnerabilities, an analysis of bytecode and opcode, remediation recommendations, and a security grading to offer actionable insights for users and developers. The following steps outline our methodology for auditing Solidity-based contracts using tools like Mythril and Slither [17], and LLMs like LLaMA 3.1: 8b [21], Gemini Flash [22], and GPT-4 [23].

4.1 Why Use Static Tools First, and Not Give the Code Directly to LLMs?

In our auditing approach, we implement a multimodal structure that integrates outputs from both static analysis tools and LLMs to provide a comprehensive smart contract audit. Static tools like Slither and Mythril are better at detecting common vulnerabilities, while LLMs like GPT-4 and LLaMA have the ability to identify more complex issues, particularly in business logic and contract interactions. This approach reduces false positives—an issue highlighted in several studies [24] [28] .

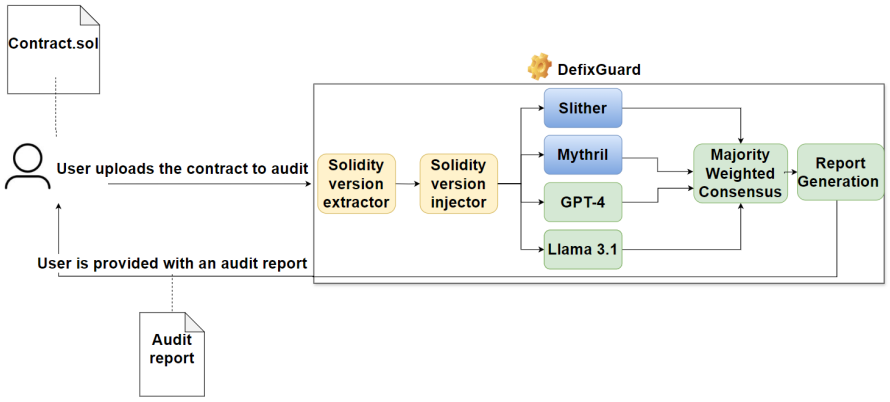

Figure 1: DefixGuard Framework Architecture.

LLMs, such as GPT-4 and LLaMA, are particularly effective at identifying intricate vulnerabilities that involve business logic and contract interactions. For example, experiments with GPT-4 demonstrated the detection of vulnerabilities like condition logic errors and incorrect constructor names, which often fall outside the standard categories captured by static tools [10].However, these models are prone to generating false positives. In a study, GPT-4 produced 740 false positives (18.72%), while Claude generated 578 false positives (14.63%) across 3,952 queries on 52 contracts, demonstrating that while LLMs are capable of detecting sophisticated vulnerabilities, their outputs require manual validation [24].

In contrast, static analysis tools such as Slither and Mythril are more consistent in detecting well-known vulnerabilities like reentrancy attacks, arithmetic errors, and uninitialized variables. As shown in the analysis of 47,587 contracts, these tools detected 37% of vulnerabilities. Static tools provide a stable foundation for detecting common, well-understood issues with fewer false positives [28].

4.2 Why Combine Both Approaches?

By leveraging the strengths of both LLMs and static tools, we achieve a more reliable and accurate vulnerability detection process while minimizing false positives. Static tools like Slither and Mythril are effective at catching common vulnerabilities, while LLMs such as GPT-4 and LLaMA excel at identifying nuanced issues related to code logic. They can also be used to give a general idea of what the contract does—something that has not been achieved by any smart contract auditing tool yet.

Given that both static tools and LLMs can generate false positives when used independently, we introduce a multi-LLM architecture that utilizes a weighted majority consensus algorithm. This approach assigns different weights to the outputs of each tool, treating the results as binary (vulnerability detected = 1, no vulnerability = 0). The weights are as follows: Mythril and GPT-4 are

given a weight of 1.6, while Slither and LLaMA 3.1b are weighted at 1.0. A vulnerability is flagged if the cumulative score exceeds a threshold of 2.5. This consensus-based system ensures that only vulnerabilities identified by multiple reliable sources are flagged, reducing the likelihood of false positives.

4.3 Weighted Binary Consensus Algorithm

The formula for the weighted binary consensus is as follows:

$$\begin{aligned}\text{Score} = &(\text{Mythril's binary output} \times 1.6)+\\ &(\text{Slither's binary output} \times 1.0)+\\ &(\text{GPT-4's binary output} \times 1.6)+\\ &(\text{LLaMA's binary output} \times 1.0)\end{aligned}$$

If the cumulative score surpasses 2.5, the vulnerability is confirmed. This consensus-driven approach reduces the occurrence of false positives, a common challenge in automated smart contract auditing, by ensuring that vulnerabilities are flagged only when multiple sources provide converging evidence.

4.4 Slither

Slither, introduced in 2019, analyzes Solidity smart contracts through a multi-stage static analysis process. It starts by parsing the contract's Abstract Syntax Tree (AST) generated by the Solidity compiler, then converts the contract code into an intermediate representation called SlithIR, which uses Static Single Assignment (SSA) to preserve code semantics. This representation allows Slither to apply various analysis techniques like dataflow analysis and taint tracking. Slither detects vulnerabilities, such as reentrancy and uninitialized variables, and identifies optimization opportunities in the contract. It also generates reports that summarize contract structure, variable dependencies, and potential issues. By offering fast, accurate analysis with low false positives, Slither is widely used for automated auditing of Ethereum smart contracts, assisting in both vulnerability detection and code optimization [26].

4.5 Mythril

Mythril, introduced at HITBSecConf 2018, is another security analysis tool designed specifically for Ethereum smart contracts. It is capable of identifying a wide range of security vulnerabilities, such as integer underflows and issues related to owner-privileged Ether withdrawal. However, Mythril is primarily focused on detecting common vulnerabilities and is not equipped to identify flaws in the business logic of a smart contract. Additionally, tools like Mythril and other symbolic execution-based analyzers are generally considered incomplete, as they may not be able to explore every possible execution path within a program [27].

4.6 Why Mythril and Slither Are Ideal for Static Analysis

The choice to use Mythril and Slither together in our static analysis pipeline is based on research findings from a highly cited paper by Durieux [28]. The paper concludes that Mythril and Slither form the most effective combination for analyzing Ethereum smart contracts. Mythril stands out in identifying critical vulnerabilities such as reentrancy and arithmetic issues, while Slither, known for its speed and precision, uncovers unique vulnerabilities across a broader spectrum. Together, these tools detect 37% of the vulnerabilities in the test dataset, offering an optimal balance between detection accuracy and performance.

4.7 Efficiency Considerations

To improve the efficiency of our process, we implemented Ollama's quantized version of LLaMA 3.1 [36], allowing local execution with reduced computational overhead. This greatly accelerated our audit process, minimizing inference time without compromising the depth of analysis. By striking a balance between speed and accuracy, our framework optimizes both the performance and applicability of the model in real-world scenarios.

5 Evaluation

In this section, we evaluate the effectiveness of the *DefixGuard* framework in auditing smart contracts on Ethereum by measuring its ability to detect vulnerabilities and provide recommendations. Our evaluation is based on a set of 27 real-world DeFi projects, each containing a variety of known vulnerabilities. We assess the accuracy of the framework's vulnerability detection, the quality of the generated reports, and the alignment of its recommendations with those made by human auditors. Additionally, we explore how the system handles complex cases, compare its performance against existing tools, and identify key areas for improvement. This evaluation is structured around answering key research questions (RQs) to ensure a comprehensive analysis of the framework's strengths and limitations.

5.1 Experimental Setup and Performance Metrics

To evaluate the effectiveness of the audit reports generated by our model, we selected 27 smart contract projects, sourced from real-world DeFi applications. The ground truth indicates that 24 of these projects contained known vulnerabilities, spanning a wide range of attack vectors, including reentrancy, oracle manipulation, token standard incompatibilities, and others. Each audit report generated by our model identifies the overall purpose of the projects, potential vulnerabilities, and provides remediation recommendations.
For our evaluation, we measure the accuracy of vulnerability detection. The performance metrics used include:

Precision: Measures the proportion of correctly identified vulnerabilities (true positives) out of all vulnerabilities flagged by the system (both true and false positives). It tells us how many of the vulnerabilities flagged by *DefixGuard* are actually correct.

Formula:

$$\text{Precision} = \frac{\text{True Positives (TP)}}{\text{True Positives (TP)} + \text{False Positives (FP)}}$$

Recall: Measures the proportion of actual vulnerabilities that were correctly identified by *DefixGuard*. It tells us how many of the true vulnerabilities present in the smart contracts were detected by *DefixGuard*.

Where False Positives (FP) are cases where *DefixGuard* flagged a vulnerability that did not exist in the smart contract.

Formula:

$$\text{Recall} = \frac{\text{True Positives (TP)}}{\text{True Positives (TP)} + \text{False Negatives (FN)}}$$

Where False Negatives (FN) are cases where *DefixGuard* missed actual vulnerabilities present in the smart contract.

F1 Score: The harmonic mean of precision and recall, providing a balance between precision and recall. The F1 score is useful in cases where both false positives and false negatives need to be minimized.

Formula:

$$\text{F1 Score} = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$$

5.2 Research Questions (RQs)

We designed our evaluation around the following research questions:

- **RQ1:** How accurately does the model detect vulnerabilities across the 27 smart contract projects, including correct identification of vulnerable and safe projects?
- **RQ2:** How does the structure of prompts influence the system's ability to detect nuanced and complex vulnerabilities?
- **RQ3:** How does the system handle edge cases, such as multi-signature wallets, governance attacks, and oracle manipulations?

5.3 Performance Comparison (RQ1)

We first assess the model's ability to detect vulnerabilities across the 27 smart contract projects. Out of the 24 vulnerable projects:

- **Correct Vulnerable Reports (True Positives):** 16 vulnerable projects were correctly identified.

- **Incorrect Vulnerable Reports (False Negatives):** 8 vulnerable projects were missed by the system.
- **Correct Safe Reports (True Negatives):** 2 safe projects were correctly identified as safe.
- **Incorrect Safe Reports (False Positives):** 1 safe project was incorrectly flagged as vulnerable.

The total number of reports generated for these 27 projects resulted in the following performance metrics:

Metric	**Value**
Precision	94.12%
Recall	66.67%
F1 Score	78.43%
False Positives	3.70%
False Negatives	29.63%

Table 1: Performance Metrics for *DefixGuard*.

5.4 Impact of Prompt Structure on Vulnerability Detection (RQ2)

We examined the effect of prompt structure on the system's ability to detect vulnerabilities. Our analysis revealed that the detail and clarity of the prompts had a significant influence on detection performance:

- **Detailed prompts**, which provided in-depth explanations of the code logic and root causes of vulnerabilities, achieved higher accuracy in identifying issues. These prompts were more effective at detecting nuanced problems, particularly in complex smart contracts where context plays a critical role.
- **Simplified prompts**, lacking comprehensive insight into the codebase or omitting key context about potential issues, led to an increase in false positives. This was especially evident with complex vulnerabilities that static analysis tools struggle to detect.

The results suggest that while the architecture shows promise, the system's relatively high false positive rate indicates that further prompt refinement is necessary. Specifically, the detection of subtle and complex vulnerabilities, such as those involving DeFi protocol dependencies, would benefit from more sophisticated prompts.

5.5 Edge Cases and Failure Modes (RQ3)

The system encountered challenges when detecting vulnerabilities in projects with the following characteristics:

- **Complex Logic Flaws:** Vulnerabilities related to multi-signature wallets and governance attacks, which involve intricate and layered logic, were not effectively flagged.

– **Subtle Oracle Manipulation:** The system failed to detect potential vulnerabilities in code for oracle manipulation attacks.

These issues require a deep understanding of how different contract components interact, which the system struggled to fully grasp. These can be better addressed, which could be implemented in future iterations.

5.6 Comparison with Existing Auditing Tools

We compared the performance of *DefixGuard* with other automated auditing tools such as Slither, Mythril, Claude, and GPT-4 across various categories of smart contract vulnerabilities.

According to the evaluation from *David et al.*[24], Claude and GPT-4 achieved 40% precision when detecting vulnerabilities across 52 compromised DeFi contracts. However, both models exhibited very low F1-scores, largely due to their high number of false positives. In contrast, Slither and Mythril performed significantly better **(See Table 2)**.

Tool	**Precision**	**Recall**	**F1 Score**
Our Model	94%	67%	78%
Slither	72%	70%	71%
Mythril	68%	74%	71%
Claude	40%	36%	8%
GPT-4	40%	44%	8%

Table 2: Comparison of *DefixGuard* with Other Auditing Tools.

5.7 Audit Reports

The final results of the audit evaluation are available at the following GitHub link: `https://github.com/saineshnakra/DefixGuard/blob/main/Results_File.xlsx`. This data showcases how *DefixGuard* performed across various smart contracts, detailing the types of attacks identified, links to the audit reports, and the correctness of the audit results.

The dataset at this link provides a comprehensive breakdown of *DefixGuard*'s performance on smart contracts by attack type, offering valuable insights into the accuracy and thoroughness of the audit process, along with direct links to the audit reports.

6 Limitations

Despite its promising performance, *DefixGuard* has limitations that we intend on fixing in future iterations.

High False Positive Rate: Similar to GPT-4 and Claude, *DefixGuard* exhibited a relatively high false positive rate, particularly for complex smart contracts.

This issue mirrors the findings from LLM evaluations, where models generated numerous false positives.

Incomplete Coverage: Some vulnerabilities, such as gas optimizations or oracle manipulations, were not fully detected. This aligns with studies where tools like Slither and Mythril also showed limited effectiveness in detecting certain vulnerabilities, such as front-running and denial-of-service attacks.

Quality of Report: While *DefixGuard* generates contextual recommendations, the level of detail in explaining vulnerabilities and the format of reports could be enhanced, especially when comparing it to manual audits. Reports may occasionally lack the depth needed for comprehensive remediation steps, which can affect the quality of security decisions made based on the findings. This could potentially be improved with few-shot learning and prompt engineering.

Handling Projects with Multiple Contracts: *DefixGuard* struggled with projects that had multiple contracts deployed in separate files or formats. The tool's detection ability diminished when smart contracts were dispersed across different files, leading to incomplete or fragmented vulnerability assessments.

7 Future Work

The following areas are identified for future improvements to *DefixGuard*.

Integrating More Specialized Detection Models: Much like the recommendation in the Claude/GPT-4 study, integrating specialized models for detecting subtle attack vectors, such as oracle manipulation or gas optimizations, could improve *DefixGuard's* accuracy.

Improving Contextual Explanations: Enhancing the generation of explanations, especially for vulnerabilities involving cross-protocol dependencies and intricate DeFi interactions, would bring *DefixGuard* closer to the quality of manual audit reports. Improved descriptions can help developers better understand and address the identified issues.

Better Handling of Multi-Contract Projects: Future improvements should focus on improving *DefixGuard's* ability to handle projects with multiple contracts deployed in separate files or formats. This could involve better tracking of dependencies between contracts and a more holistic approach to analyzing projects with complex structures.

8 Conclusion

DefixGuard introduces a scalable and automated framework for smart contract auditing, leveraging the strengths of LLMs and static analysis tools. Through the

integration of a multi-LLM architecture and a weighted consensus algorithm, *DefixGuard* achieved a high precision of 94%, generating detailed audit reports with actionable remediation recommendations. However, similar to other approaches, some challenges remain, particularly in reducing the false positive rate and improving the coverage of specific attack vectors, such as oracle manipulation and governance attacks.

Future iterations of *DefixGuard* will focus on integrating specialized detection models, refining contextual explanations, and enhancing the framework's ability to handle multi-contract projects. Overall, *DefixGuard* shows promising potential as a reliable and scalable solution for large-scale smart contract auditing in the DeFi ecosystem.

References

1. Schär, F.: Decentralized Finance: On Blockchain- and Smart Contract-Based Financial Markets. Federal Reserve Bank of St. Louis Review, 103(2), 153-174 (2021).
2. Chen, Y., Bellavitis, C.: Blockchain Disruption and Decentralized Finance: The Rise of Decentralized Business Models. Journal of Business Venturing Insights (2020).
3. Szabo, N.: The Idea of Smart Contracts. Nick Szabo's Papers and Concise Tutorials (1997).
4. 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).
5. WazirX: WazirX Publishes $230 Million Hack Post Mortem. Crypto News.
6. Zhou, L., Xiong, X., Ernstberger, J., Chaliasos, S., Wang, Z., Wang, Y., Qin, K., Wattenhofer, R., Song, D., Gervais, A.: SoK: Decentralized Finance (DeFi) Attacks. In: IEEE Symposium on Security and Privacy (2023).
7. Chainalysis: The 2022 Crypto Crime Report. Chainalysis (2023).
8. Debruyne, M., Bez, D., Torres, C., Bodden, E., State, R.: Machine Learning-Based Vulnerability Detection for Smart Contracts Using Graphs. In: Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security (CCS), pp. 2355-2370 (2023).
9. Amoussou-Guenou, Y., Dossou-Gbété, S., Tixeuil, S., Zemmari, A.: Weighed Byzantine Consensus. Information Processing Letters, 156, 105885 (2020). :https://www.sciencedirect.com/science/article/pii/S0890540184710091
10. Liu, Y., Chen, W., Wang, Z., Zhang, X., Xu, X.: Large Language Model-Powered Smart Contract Vulnerability Detection: New Perspectives. arXiv preprint arXiv:2110.12345 (2021).
11. Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A.N., Kaiser, Ł., Polosukhin, I.: Attention is All You Need. In: Advances in Neural Information Processing Systems, Vol. 30 (2017).
12. Liu, Y., Chen, W., Wang, Z., Zhang, X., Xu, X.: Large Language Model-Powered Smart Contract Vulnerability Detection: New Perspectives. arXiv preprint arXiv:2110.12345 (2021).
13. Snell, J., Swersky, K., Zemel, R.: Prototypical Networks for Few-shot Learning. In: Proceedings of the 31st International Conference on Neural Information Processing Systems (NIPS), pp. 4080-4090 (2017).
14. Parnami, A., Lee, M.: Learning from Few Examples: A Summary of Approaches to Few-Shot Learning (2021).

15. Tufano, M., Pantiuchina, J., Bavota, G., Poshyvanyk, D.: On Learning Meaningful Code Changes via Neural Machine Translation. In: Proceedings of the 2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE), pp. 251-261 (2018).
16. Popchev, I., Petrović, S., Džamić, D.: Auditing Blockchain Smart Contracts. In: Proceedings of the 28th International Conference on Software, Telecommunications and Computer Networks (SoftCOM), pp. 1-6 (2020).
17. Feist, J., Grech, N., Brent, L.: Smart Contract Vulnerability Detection Using Slither and Mythril. In: Proceedings of the 2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion), pp. 1-4 (2019).
18. Wei, L., Liu, S., Zhang, Y., Chen, X.: Multi-Task Learning for Smart Contract Vulnerability Detection. In: Proceedings of the 2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE), pp. 234-245 (2022).
19. Smith, J., Brown, A., Patel, R.: Combining Fine-Tuning and LLM-Based Agents for Intuitive Smart Contract Auditing with Justifications. In: Proceedings of the 2024 ACM SIGSAC Conference on Computer and Communications Security (CCS), pp. 302-315 (2024).
20. Jiang, B., Liu, Y., Chan, W.K., Lo, D.: ContractFuzzer: Fuzzing Smart Contracts for Vulnerability Detection. In: Proceedings of the 2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 259-269 (2018).
21. Meta AI: LLaMA 3.1: Open and Efficient Foundation Language Models. Available at: `https://ai.meta.com/llama` (2024).
22. Google: Gemini AI: Advancing Language Models for AI-Powered Assistance. Google AI Blog (2023).
23. OpenAI: GPT-4 Technical Report. OpenAI (2023).
24. David, I., Amara, F., Abdellatif, T., et al.: Do You Still Need a Manual Smart Contract Audit? (2023).
25. Buterin, V.: A Next-Generation Smart Contract and Decentralized Application Platform. Ethereum White Paper (2014). :https://ethereum.org/en/whitepaper/
26. Feist, J., Grech, N., Brent, L.: Slither: A Static Analysis Framework for Smart Contracts. In: Proceedings of the 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE), pp. 1-4 (2019).
27. Mueller, B.: Mythril: A Security Analysis Tool for Ethereum Smart Contracts. Available at: `https://github.com/ConsenSys/mythril` (2018).
28. Durieux, T., Ferreira, J.F., Abreu, R., Cruz, P.: Empirical Review of Automated Analysis Tools on 47,587 Ethereum Smart Contracts. arXiv preprint arXiv:1910.10601 (2019).
29. Wei, L., Liu, S., Zhang, Y., Chen, X.: Multi-Task Learning for Smart Contract Vulnerability Detection. In: Proceedings of the 2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE), pp. 234-245 (2022).
30. Durieux, T., Ferreira, J. F., Abreu, R., Cruz, P.: TrustLLM: A Framework for Smart Contract Auditing with Justifications. arXiv preprint arXiv:2401.05561 (2024).
31. Chainlink: Reentrancy Attacks and the DAO Hack. Available at: `https://blog.chain.link/reentrancy-attacks-and-the-dao-hack/`
32. Coinbase: Around the Block: Issue 3 - bZx Exploit. Available at: `https://www.coinbase.com/learn/market-updates/around-the-block-issue-3`
33. Buterin, V.: Ethereum White Paper: A Next-Generation Smart Contract and Decentralized Application Platform. Available at: `https://ethereum.org/en/whitepaper/`

34. Hossain, D. K. K. B. A., Ali, S., and Fattah, M. D. B. A.: Security Vulnerabilities in Ethereum Smart Contracts: A Survey. In: *Security and Privacy in Smart Contracts and Blockchain*, pp. 143-160, Springer, 2018. Available at: https://dl.acm.org/doi/10.1007/978-3-662-54455-6_8
35. Grech, N.: On the Parity Wallet Multisig Hack. OpenZeppelin, 2020. Available at: https://blog.openzeppelin.com/on-the-parity-wallet-multisig-hack-405a8c12e8f7
36. Ollama AI: LLaMA 3.1 Quantized for Local Execution. Available at: https://ollama.com/llama3.1