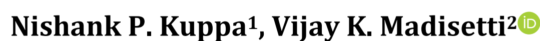
Scientific Research Publishing

Journal of Information Security, 2025, 16(1), 197-226
https://www.scirp.org/journal/jis
ISSN Online: 2153-1242
ISSN Print: 2153-1234

Robust Detection and Analysis of Smart Contract Vulnerabilities with Large Language Model Agents


Nishank P. Kuppa[1], Vijay K. Madisetti[2]

[1]College of Computing, Georgia Institute of Technology, Atlanta, GA, USA
[2]School of Cybersecurity and Privacy, Georgia Institute of Technology, Atlanta, GA, USA
Email: nkuppa3@gatech.edu, vkm@gatech.edu

How to cite this paper: Kuppa, N.P. and Madisetti, V.K. (2025) Robust Detection and Analysis of Smart Contract Vulnerabilities with Large Language Model Agents. *Journal of Information Security*, **16**, 197-226.
https://doi.org/10.4236/jis.2025.161011

Received: December 17, 2024
Accepted: January 23, 2025
Published: January 26, 2025

Copyright © 2025 by author(s) and Scientific Research Publishing Inc.
This work is licensed under the Creative Commons Attribution International License (CC BY 4.0).
http://creativecommons.org/licenses/by/4.0/

Open Access

Abstract

Smart contracts on the Ethereum blockchain continue to revolutionize decentralized applications (dApps) by allowing for self-executing agreements. However, bad actors have continuously found ways to exploit smart contracts for personal financial gain, which undermines the integrity of the Ethereum blockchain. This paper proposes a computer program called SADA (Static and Dynamic Analyzer), a novel approach to smart contract vulnerability detection using multiple Large Language Model (LLM) agents to analyze and flag suspicious Solidity code for Ethereum smart contracts. SADA not only improves upon existing vulnerability detection methods but also paves the way for more secure smart contract development practices in the rapidly evolving blockchain ecosystem.

Keywords

Blockchain, Ethereum, Smart Contracts, Security, Decentralized Applications, Web3, Cryptocurrency, Large Language Models

1. Introduction

According to IBM, a smart contract is a digital contract stored on a blockchain (such as Ethereum) that automatically executes when certain predefined terms and conditions are satisfied [1]. However, the rapid adoption of Ethereum smart contracts has outpaced the development of robust security measures. This project therefore aims to create a more sophisticated, adaptive, and accurate method of vulnerability detection that can keep up with the evolving complexity of smart contracts and the ingenuity of potential attackers.

This project leverages multiple Large Language Model (LLM) agents to analyze smart contracts, with a particular focus on identifying patterns of suspicious or potentially malicious code. The use of LLM agents, such as those built on GPT-4o, offers a powerful new approach to vulnerability detection by enabling these agents to understand and process code contextually. Rather than relying on static code analysis, which often struggles to keep up with rapidly evolving attack vectors and is not always up to date, the deployment of LLM agents allows for a more comprehensive form of analysis. These agents can recognize common yet subtle vulnerabilities—such as reentrancy issues, integer overflows, and unchecked external calls—that are frequently exploited in decentralized applications and smart contracts [2]. For instance, LLM agents can flag outdated functions associated with historical vulnerabilities, such as the unsafe and strongly discouraged use of tx.origin [2]. Additionally, they can identify the use of older Solidity pragmas or detect attempts to manipulate contract states in ways that compromise integrity or security [2].

The motivation for this project largely stems from The DAO incident in June 2016, which was a pivotal event in Ethereum's history that exposed critical vulnerabilities in smart contracts. The DAO was a decentralized venture capital fund that raised approximately 150 million USD worth of Ether through a token sale. However, an attacker exploited a vulnerability in The DAO's smart contract code, specifically a reentrancy flaw, to drain about 60 million USD worth of Ether into a separate DAO [3]. This attack led to Ethereum implementing a hard fork to reverse the theft, effectively creating two separate blockchains: Ethereum (ETH) and Ethereum Classic (ETC). This event underscores the need for advanced, proactive smart contract vulnerability detection agents such as the one proposed for this project.

As smart contracts become more popular and Web3 applications proliferate, proactive vulnerability detection will become more crucial than ever. This project aims to provide an adaptable and intelligent defense mechanism capable of identifying and mitigating security risks before contracts are deployed, thus contributing to a more secure and resilient blockchain environment. This document is structured as described below:

- Section 2 describes existing work in the field of smart contract vulnerability detection.
- Section 3 provides the reader with relevant background information.
- Section 4 describes the proposed approach.
- Section 5 describes the test plan for validating the proposed approach.
- Section 6 provides the results of the test plan.
- Section 7 provides further analysis and comparison with prior work.
- Section 8 explains challenges and limitations with the proposed approach.
- Section 9 presents conclusions and recommendations for future work.

2. Existing Work

This section presents prior approaches that this paper aims to improve upon.

2.1. Non-LLM Based Tools

The field of smart contract vulnerability detection has seen significant development in traditional automated analysis tools prior to the development of LLM-based solutions. Leading the early efforts, in 2016, Luu *et al.* described a tool called Oyente [4], which pioneered the use of symbolic execution to identify common vulnerabilities in Ethereum smart contracts. By analyzing control flow graphs and exploring possible execution paths, Oyente established fundamental techniques for detecting issues like reentrancy, timestamp dependence, and transaction ordering dependencies [4]. However, its high false-positive rate and limited scope led to the need for more sophisticated approaches.

Another tool called Mythril was introduced at the Hack in the Box Security Conference in 2018 and emerged as a comprehensive security analysis tool that combines multiple analysis methods. It uses dynamic symbolic execution, taint analysis, and control flow checking to identify a broader range of vulnerabilities [5]. Mythril's ability to detect complex attack vectors and provide detailed exploit scenarios made it a valuable tool for smart contract auditors, though its computational intensity and complex setup requirements present practical limitations.

Manticore, which was introduced in 2019, is another commonly used smart contract vulnerability detection tool that leverages dynamic symbolic execution and concrete execution capabilities, which allows developers to simulate contract execution under various conditions [6]. However, like other tools, its scalability limitations and complex setup requirements restrict its widespread usage as well as practical application in larger projects.

One popular and widely used tool that was introduced in 2019 is Slither, which performs rapid vulnerability detection by running a suite of vulnerability detectors [7]. Its speed and relatively low false-positive rate make it particularly suitable for continuous integration pipelines, though it remains limited to detecting known vulnerability patterns and cannot identify novel attack vectors. It also cannot detect vulnerabilities that typically show up during contract execution, such as arithmetic-related issues.

2.2. LLM Based Tools

There has been substantial research and attempts to use LLM agents to improve vulnerability detection in smart contracts, though they mostly deal with static analysis that involves a singular LLM. Boi *et al.* explore training and fine-tuning a model called Llama-2-7b-chat-hf [8]. The research found that the fine-tuned model correctly identified malicious smart contracts 59.5% of the time [8].

Another paper by He *et al.* uses graphs to identify timestamp vulnerabilities, reentrancy vulnerabilities, and access control vulnerabilities in smart contracts [9]. He *et al.* indicate that their model achieves an accuracy of 84.63%, 92.55%, and 61.36% respectively, which demonstrates that their method surpasses other smart contract vulnerability detection techniques, such as static analysis [9].

A study by Ma *et al.* explores the use of multiple LLM agents for analyzing smart

contract code by dividing the static analysis process among several specialized agents [10]. However, their agent architecture differs from the one proposed in this paper. Despite these differences, their work provides a valuable foundation for this research.

3. Background

This section presents relevant background information to help the reader understand the scope of this project.

3.1. Definition of a Smart Contract

A smart contract is a self-executing program that automates the actions required in a blockchain transaction [11]. It is essentially a digital agreement stored on a blockchain network that automatically executes when predetermined terms and conditions are met [11]. Smart contracts are written in blockchain-specific programming languages, the most common being Solidity for Ethereum. These digital contracts operate without the need for intermediaries, which allows for increased efficiency, transparency, and security in various transactions and processes.

Despite the widespread use of smart contracts, many limitations still exist. Smart contracts rely heavily on the accuracy of the initial programming, and any errors or loopholes in the code can lead to unintended consequences, such as The DAO incident in 2016. Additionally, smart contracts are immutable, meaning once a smart contract is deployed, it cannot be changed or updated. Oftentimes, vulnerabilities are present in smart contracts, and they can occur either through inadvertent human error during development or due to nefarious intent. Therefore, there is a need for resilient and comprehensive tools that can detect vulnerabilities in smart contracts to protect all relevant parties involved in a smart contract and therefore prevent financial losses.

3.2. OWASP Top 10

In order to determine the most important vulnerabilities to focus on, the OWASP (Open Worldwide Application Security Project) Top 10 for smart contracts is used as a reference. Below is a description of the top 10 vulnerabilities from OWASP as of 2023.

1) **Reentrancy Attacks:** Occurs when a function makes an external call before updating its state, allowing malicious contracts to reenter and repeat actions like withdrawals [12]. To mitigate this, it is recommended to implement the Checks-Effects-Interactions (CEI) pattern and use reentrancy guards such as OpenZeppelin, to ensure state updates occur before external calls.

2) **Integer Overflow and Underflow:** Arithmetic operations that exceed the maximum or minimum value of a variable type, potentially leading to unexpected behavior [12]. Developers can mitigate this risk by using Safe-Math libraries or leveraging Solidity version 0.8.0 and above, which include built-in overflow

checks.

3) **Timestamp Dependence:** Relying on block timestamps for critical operations, which can be manipulated by miners within certain bounds [12]. To address this, developers should use block numbers instead of timestamps for time-dependent logic or implement a time buffer to reduce the impact of minor timestamp manipulations.

4) **Access Control Vulnerabilities:** Failure to properly restrict access to sensitive functions or data, allowing unauthorized users to perform critical operations [12]. To address this, developers should utilize robust access control mechanisms using modifiers and role-based access control (RBAC).

5) **Front-Running Attacks:** This involves exploiting knowledge of pending transactions to gain unfair advantages by placing transactions with higher gas fees [12]. Developers can counter this by implementing commit-reveal schemes or using private mempools (a temporary storage area for unconfirmed transactions on a blockchain) for sensitive transactions [13].

6) **Denial of Service (DoS) Attacks:** Targeting vulnerabilities to exhaust critical resources, rendering contracts non-functional [12]. To mitigate these attacks, developers should implement gas limits, avoid loops with unbounded length, and use pull payment systems instead of push systems, ensuring the contract remains operational under various conditions.

7) **Logic Errors:** Subtle flaws in contract logic that deviate from intended behavior, potentially leading to exploitable conditions [12]. Thorough code reviews, comprehensive unit tests, and formal verification of critical functions can help to identify and eliminate these errors before deployment.

8) **Insecure Randomness:** Challenges in generating true randomness on deterministic blockchain networks, potentially allowing prediction or manipulation of random numbers [12]. Using external sources of randomness like Chainlink VRF can provide more secure randomness generation.

9) **Gas Limit Vulnerabilities:** Functions exceeding block gas limits, particularly those involving loops over dynamic data structures, risking transaction failure [12]. Developers should optimize gas usage, implement gas-efficient algorithms, and avoid unbounded loops to ensure transactions can be processed within gas limits.

10) **Unchecked External Calls:** Failing to verify the outcome of external function calls, potentially compromising contract integrity and functionality [12]. Always checking the return values of external calls and using the transfer() or send() functions for Ether transfers, which automatically revert on failure, can prevent issues arising from failed external interactions.

3.3. SWC Registry

Another popular resource for smart contract developers is the SWC (Smart Contract Weakness Classification) Registry. This registry was created by the Consensys Diligence and contains 37 common vulnerability patterns. **Table 1** showcases ten of the most relevant and common vulnerabilities for this research [2] [8].

Table 1. Vulnerabilities from the SWC registry.

SWC ID	Name	Description
SWC-101	Integer Overflow and Underflow	Arithmetic operations can exceed the maximum or minimum value, causing unwanted behavior
SWC-104	Unchecked Call Return Value	Failure to check return values from external calls may lead to unexpected states
SWC-105	Unprotected Ether Withdrawal	Failing to restrict access to withdrawal functions may allow unauthorized access to funds
SWC-106	Unprotected SELFDESTRUCT Instruction	Lack of restrictions allows any user to destroy the contract and reclaim funds
SWC-107	Reentrancy	Calling external contracts can allow malicious actors to re-enter the calling contract
SWC-109	Uninitialized Storage Pointer	Using storage variables without initialization can lead to unpredictable states
SWC-112	Delegatecall to Untrusted Callee	Using delegatecall with untrusted addresses may allow malicious code execution
SWC-113	DoS with Failed Call	Contracts can have loops that may run out of gas, leading to service denial
SWC-116	Block values as a proxy for time (Timestamp Dependence)	Relying on block timestamps for critical logic can lead to manipulation by miners
SWC-120	Weak Sources of Randomness from Chain Attributes	Using non-secure sources for randomness can be exploited for predictability

4. SADA: Our Proposed Approach

This section presents SADA's overall design as well as its key components, including how they work together.

4.1. Overview

SADA is designed with multiple distinct roles and interactions for each agent. At a high level, the program operates as follows:

1) **Input and Validation:** Users provide a smart contract's address. SADA ensures the address is in a valid checksum format, using the Web3 library to convert it if necessary.

2) **Source Code Retrieval:** SADA fetches the source code of the smart contract from Etherscan, using the provided address as a query.

3) **Static Analysis:** The static analysis agent uses a fine-tuned GPT-4o model to examine the source code for vulnerabilities, patterns, and compliance with best practices. It specializes in identifying issues without executing the code.

4) **Dynamic Analysis:** The dynamic analysis agent simulates contract execution using the Web3 Python library to uncover runtime-specific behaviors and vulnerabilities, and then uses GPT-4o to analyze the simulation results.

5) **Synthesis of Results:** An agent powered by GPT-4o consolidates the findings from the static and dynamic analyses, with the goal of providing the user with a unified, detailed report of the contract's security posture.

It is important to note that for the purposes of this project, smart contracts that have already been deployed are being evaluated, which is why contract addresses

are used as input.

The dataset used for training and testing comes from the Slither Audited Smart Contracts dataset, which is available on Hugging Face [14]. Eleven smart contracts from this dataset were used for training, and another eleven were used for validation.

4.2. Multi-Agent Advantage

The SADA framework incorporates a multi-agent architecture to enhance the precision, scalability, and depth of smart contract vulnerability detection. This approach divides the analysis workload among specialized agents, each tailored to a specific aspect of the analysis process. The use of multiple agents offers several distinct advantages over a monolithic analysis framework.

Each agent in SADA focuses on a unique task, such as static analysis, dynamic analysis, or synthesizing the information generated from each analysis. More specifically, the static analysis agent excels at identifying code-level vulnerabilities by parsing Solidity contracts and detecting known patterns like reentrancy or unchecked external calls. On the other hand, the dynamic analysis agent connects to the Ethereum blockchain to simulate contract execution and identify vulnerabilities that only manifest during this process. Attempting to combine all of these specialized tasks into one agent would lead to more missed vulnerabilities due to limited context window and model overhead. Also, dividing tasks into individual agents offers a layer of redundancy that mitigates errors and improves overall system reliability.

5. Model

5.1. Dataset

The Slither Audited Smart Contracts dataset is used for training, testing, and validation. It comprises several thousand Solidity contracts sourced from Etherscan.io. This dataset offers a unique combination of contract addresses, source code, deployed bytecode, and comprehensive vulnerability classifications derived from the Slither static analysis framework, making it an indispensable tool for security research and development.

The labeling system of the vulnerabilities is particularly sophisticated. It encompasses 38 distinct vulnerability classes that are thoughtfully consolidated into nine primary categories, with a specific designation (label 4) reserved for contracts deemed secure.

5.2. Static Analysis Agent

Static analysis is a method of vulnerability detection in which a program's code structure is analyzed to look for vulnerability patterns, without executing the code [15]. Benefits of static analysis include that it can be performed anytime during the development process, it can detect a wide variety of problems, and it can be easily automated [15]. Examples of issues that static analysis excels at detecting

include uninitialized variables, improper visibility modifiers, and reentrancy risks.

The static analysis agent in SADA uses a fine-tuned OpenAI GPT-4o agent to scrutinize the source code of Ethereum smart contracts and effectively perform static analysis. This agent is fine-tuned using a JSONL file containing eleven entries. Each entry contains a system prompt, a pre-selected smart contract with vulnerabilities taken from the Slither Audited Smart Contracts dataset, and a corresponding report detailing the vulnerabilities in each smart contract and relevant solutions. The report is organized in the desired output so that the model can learn the appropriate response style.

Careful prompt engineering in the system prompt is needed in order to instruct GPT-4o on the structure and level of detail needed in its response. The system prompt used in SADA can be found in Appendix 11.1.

The vulnerabilities used to fine-tune GPT-4o are chosen based on the most common vulnerabilities listed on the SWC Registry as well as the OWASP Top 10. A breakdown of the vulnerabilities in each smart contract in the training data can be found in Table 2. Relevant vulnerabilities from the OWASP Top 10 and SWC Registry are incorporated into the fine-tuning dataset through a series of smart contracts that exemplify these risks. Each smart contract included in the training data is paired with a descriptive vulnerability report, which helps the model understand the nuances of each issue and the appropriate mitigation strategies.

Table 2. Vulnerabilities in smart contracts used for training.

Contract	Vulnerabilities
Contract 1	Locked Ether, Reentrancy, Unchecked "burn" Function, Race Condition, Gas Inefficiency, Locked Tokens Potential
Contract 2	Locked Ether, Delegatecall to Untrusted Contract, Missing Event Emission, Unrestricted Ether Reception, Proxy Hijacking Potential, Gas Inefficiency
Contract 3	Reentrancy, Locked Ether, Race Condition Potential, Gas Inefficiency, Unchecked External Call Returns, Missing Event Emission, Integer Overflow Potential
Contract 4	Incorrect Inequality Check, Locked Ether, Reentrancy, Unchecked Returns, Integer Overflow Potential, Missing Event Emission, Gas Inefficiency, Deprecated Functions
Contract 5	Uninitialized State Variables, Weak PRNG, Incorrect Equality Check, Uninitialized Local Variables, Gas Inefficiency, Unchecked Returns, Missing Event Emission, Deprecated Functions
Contract 6	Weak PRNG, Timestamp Dependence, Incorrect Equality Check, Reentrancy, Missing Event Emission, Locked Ether, Gas Inefficiency, Deprecated Functions, Function Visibility Issues
Contract 7	Reentrancy, Race Condition Potential, Unchecked "burn" Function, Missing Event Emission, Frozen Tokens Potential, Locked Ether, Gas Inefficiency, Deprecated Functions
Contract 8	Reentrancy, Locked Ether, Unchecked External Call Returns, Integer Overflow Potential, Missing Event Emission, Gas Inefficiency, Price Manipulation Potential, Missing Access Control
Contract 9	Weak PRNG, Front-Running Potential, Reentrancy, Locked Ether, Unchecked Returns, Integer Overflow Potential, Missing Event Emission, Denial of Service Potential, Gas Inefficiency, Function Visibility Issues
Contract 10	Reentrancy, Race Condition Potential, Unchecked Returns, Missing Event Emission, Gas Inefficiency, Missing Access Control, Deprecated Functions
Contract 11	Reentrancy, Weak PRNG, Locked Ether, Unchecked Returns, Integer Overflow Potential, Gas Inefficiency, Missing Event Emission

For the fine-tuning process, the model is trained on 675,369 tokens across 9 epochs, using a batch size of 1 and a learning rate multiplier of 2. The parameters were automatically selected by the system. After completing the training, the model achieved a low training loss of 0.0063, indicating that it effectively learned the patterns of smart contract vulnerabilities. Figure 1 shows the loss curve that was generated by the fine-tuning process over 99 steps. This fine-tuning process properly equips the GPT-4o model with the expertise to perform more reliable static analysis compared to using the standard model.

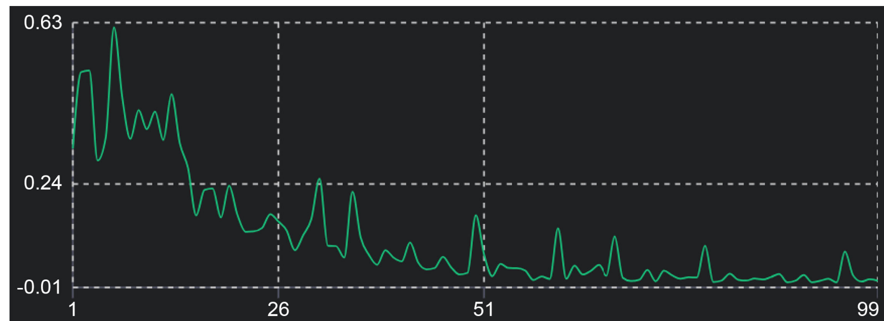


Figure 1. Loss curve from fine-tuning GPT-4o.

Other parameters mostly remained at their default values, including Top P at 1, frequency penalty at 0, and presence penalty at 0. However, we adjusted the temperature parameter from 1 to 0.7 to minimize randomness and reduce the chance of hallucination by the LLM.

With this fine-tuned model, the static analysis agent in SADA can scan Ethereum smart contracts for common security weaknesses, providing developers with a powerful tool to enhance the security of their blockchain applications. The result is a more reliable, efficient, and secure smart contract analysis pipeline that assists developers in addressing vulnerabilities proactively before they become exploitative risks.

An example of a report from the static analysis agent can be found in Appendix 11.2.

5.3. Dynamic Analysis Agent

Dynamic analysis is a method of vulnerability detection in which issues are identified during program execution [15]. These are problems that static analysis might miss because they only manifest when the program is executed. Benefits of dynamic analysis include the ability to identify vulnerabilities that are not easy to identify in static analysis, the ability to test software in a realistic environment, and the ability to focus more on runtime behavior and performance optimization [15]. It can also detect timing-related attacks, assess gas consumption patterns, validate access control mechanisms, and ensure that contract functions behave consistently under different computational conditions.

The dynamic analysis agent in SADA consists of two key components: the Web3 Python library [16], which simulates the execution of the contract based on

default parameters, and an instance of GPT-4o which analyzes the results of the contract simulation. Initially, the agent establishes a connection to the Ethereum blockchain through Web3, using a private Infura HTTP provider URL. The agent then checks whether a given address is a smart contract by inspecting the code stored at that address; if code is found, it confirms the address as a smart contract.

The agent then retrieves the contract's ABI (Application Binary Interface) via the Etherscan API. The ABI is crucial for interacting with the contract, as it defines the functions and data types available. The ABI is also essential in allowing the Web3 library to understand how the contract code is set up for instantiation purposes [17].

Subsequently, the agent tests a selection of common ERC20 token functions such as name, symbol, decimals, totalSupply, balanceOf, and allowance, checking whether the contract behaves as expected and returns correct results. If a function call fails, the agent logs the error. Additionally, it dynamically tests all functions defined in the contract's ABI, using mock parameters based on the expected input types. This ensures comprehensive testing of the contract's behavior and logs any errors encountered during execution.

The analysis component of the agent generates a detailed report based on the ABI and test results. This report includes basic function test results as well as more advanced contract functionality tests, emphasizing potential issues like token transfer vulnerabilities or improper permissions. The results are then passed to the same fine-tuned GPT model used in the static analysis agent for deeper security analysis, which assesses the contract for slippage protection, front-running risks, reentrancy risks, and other common vulnerabilities. The GPT model also provides actionable recommendations, such as improving access control, mitigating known exploits, and optimizing the contract for gas efficiency and security. Specific examples of vulnerable pieces of code from the contract, as well as recommended fixes, are also provided.

An example of a report from the dynamic analysis agent can be found in Appendix 11.3.

5.4. Orchestration Layer

An orchestration layer is essential to manage the workflow of SADA efficiently. This layer is responsible for accepting the user-inputted contract address, retrieving the relevant information, sending it to both the static and dynamic analysis agents, and synthesizing the results at the end.

The orchestration layer operates as a separate Python program that follows the workflow as shown in **Figure 2**. It begins by retrieving the contract's source code through the Etherscan API. This code is then analyzed by the static analysis agent, and the resulting static analysis report is generated and presented to the user. Subsequently, the agent performs dynamic analysis by simulating the execution of the smart contract on the Ethereum blockchain, and then observing the contract's behavior during runtime. The raw dynamic analysis report is refined using GPT,

resulting in a detailed and easy-to-understand report that shows identified vulnerabilities and associated risks.

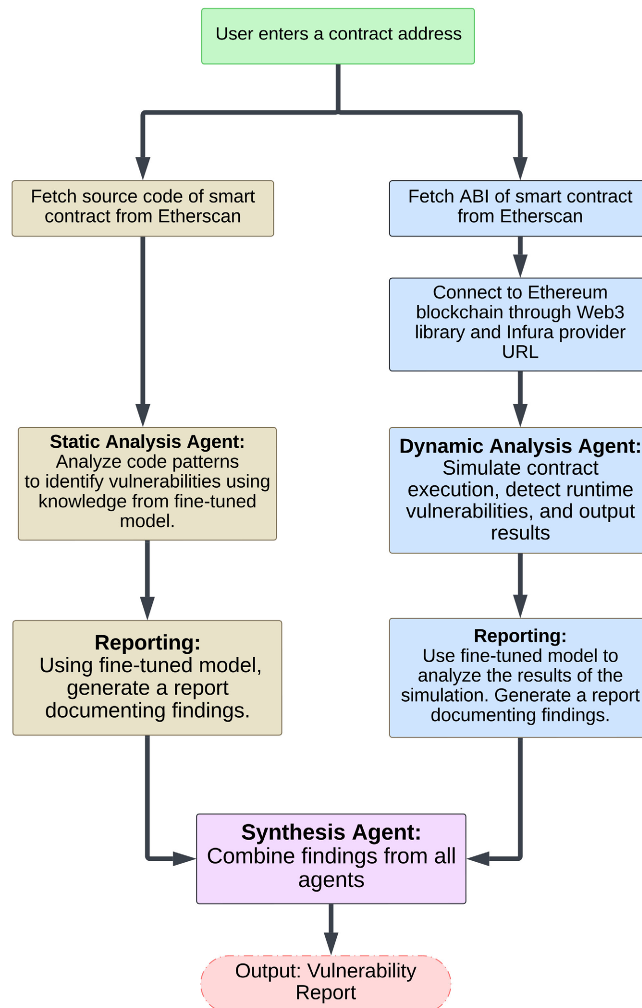


Figure 2. Orchestration layer.

Once both static and dynamic analysis reports are completed, the synthesis agent—utilizing a fresh instance of GPT rather than the fine-tuned model—combines the findings into a comprehensive report. This report evaluates the severity of each vulnerability, assesses the potential for exploits, and provides actionable recommendations for mitigating the risks. It also suggests improvements for the contract's overall security based on best practices in Solidity development.

Finally, the synthesized report is saved as a markdown file and presented to the user, offering a thorough overview of the smart contract's security posture. This process helps streamline the identification and mitigation of potential risks, and provides developers with clear and actionable insights with specific code examples to secure their smart contracts effectively. Figure 3 shows a snapshot of an example vulnerability report. The report clearly describes the issue, risk, severity, exploitation path, and mitigation suggestions for each vulnerability.

An example of a synthesized report can be found in Appendix 11.4.

Synthesis of Static and Dynamic Analysis Findings

Identified Vulnerabilities and Recommendations

1. Reentrancy Vulnerability

- **Description**: Functions such as `transfer`, `transferFrom`, `multiPartyTransfer`, and `multiPartyTransferFrom` are vulnerable to reentrancy attacks as they interact with external contracts without proper reentrancy protection.
- **Risk**: An attacker could exploit this to repeatedly call these functions and manipulate token balances during reentrant calls, leading to unauthorized transfers.
- **Severity**: Critical
- **Exploitation**: An attacker could use a malicious contract to repeatedly call vulnerable functions before the state is updated, draining tokens.
- **Mitigation**:
  - Apply the *Checks-Effects-Interactions* pattern.
  - Use the `ReentrancyGuard` from OpenZeppelin to lock functions during execution.

```
import "\@openzeppelin/contracts/security/ReentrancyGuard.sol";

contract Usdcoins is ReentrancyGuard {
    function transfer(address _to, uint256 _value) public nonReentrant
    returns (bool success) {
        require(_to != address(0));
        balanceOf[msg.sender] = balanceOf[msg.sender].sub(_value);
        balanceOf[_to] = balanceOf[_to].add(_value);
        emit Transfer(msg.sender, _to, _value);
        return true;
    }
}
```

Figure 3. Synthesis of static and dynamic analysis findings: Reentrancy vulnerability.

5.5. Model Choice

Selecting an appropriate language model for analyzing smart contract vulnerabilities was a critical decision in this project. After performing an evaluation of available LLMs (namely Anthropic Claude, Meta Llama, and OpenAI GPT), GPT-4o is chosen as the optimal foundation due to its exceptional capabilities and performance advantages in smart contract analysis.

One of the key advantages of GPT-4o is that it is hosted by OpenAI, which simplifies the fine-tuning process. Unlike open source models such as Meta Llama that require local hosting, GPT-4o eliminates the need for complex setup. This not only reduces the technical overhead but also accelerates the deployment process. OpenAI also provides a user-friendly web interface for developers that makes the fine-tuning process straightforward and accessible, even for those who may not have extensive technical expertise.

Initially, GPT-4o mini was selected for its efficiency and cost-effectiveness. However, during preliminary testing, it became evident that the model had a high false positive rate of approximately 15%. This level of inaccuracy was deemed unacceptable, and as result, it was necessary to seek a more reliable model, such as GPT-4o, to ensure the accuracy of the vulnerability analysis, despite it being more expensive.

6. Testing

To evaluate the performance and effectiveness of SADA, both qualitative and quantitative assessments of the synthesized reports are conducted. This section

discusses the specifics of both assessments.

6.1. Testing Overview

To manage the scope of the evaluation, the assessments presented here primarily compare SADA against two commonly used tools for smart contract analysis, namely Slither and ChatGPT. Performing an exhaustive evaluation of SADA against a broad range of smart contract analysis tools would require substantial additional effort beyond the scope of this work. Each tool operates using different methodologies, detection strategies, and environments, which would necessitate detailed configuration, specialized setup, and manual validation of results for every vulnerability class.

Despite these challenges, Section 8 provides a comprehensive comparison of SADA against a selection of popular and widely recognized tools in the smart contract analysis domain.

6.2. Quantitative Assessment

The quantitative assessment focuses on vulnerability detection accuracy, which is the core function of SADA. This evaluation uses a subset of the Slither Audited Smart Contracts dataset, which contains smart contracts with labeled vulnerabilities identified by Slither. Slither is a widely used and trusted static analysis tool in blockchain security [7] [14]. The validation dataset includes contract addresses, smart contract code, and the corresponding vulnerabilities, providing a reliable comparison for the evaluation. For the purposes of evaluation, eleven smart contracts with various amounts of vulnerabilities are used.

For each contract in the validation dataset, the vulnerabilities labeled by Slither serve as a reference point. Both SADA and ChatGPT, the latter also used as a baseline for comparison, analyze the same smart contracts to identify vulnerabilities. The model used by ChatGPT is GPT-4o, with no additional prompting other than asking it to identify a given smart contract for vulnerabilities. The performance of each tool is assessed by measuring the following metrics:

- **True positives (TP):** Correctly identified vulnerabilities.
- **False positives (FP):** Incorrectly flagged vulnerabilities.
- **False negatives (FN):** Missed vulnerabilities.

These metrics enable the calculation of precision and recall, where precision measures the percentage of flagged vulnerabilities that are true positives, and recall measures the percentage of actual vulnerabilities correctly identified by the tool [18]. The F1 score of each tool is also calculated based on the precision and recall values.

The evaluation highlights instances where SADA identifies vulnerabilities that Slither misses. These cases, where SADA's dynamic analysis uncovers runtime vulnerabilities overlooked by Slither's static analysis, are counted as true positives for SADA. A similar approach is applied to ChatGPT. This allows the evaluation to emphasize SADA's strengths, particularly in detecting issues that static analysis

alone may miss. Additionally, vulnerabilities missed by both Slither and SADA, or by Slither and ChatGPT, under-score the limitations of automated tools and the importance of using complementary approaches. These cases are treated as false negatives for both SADA and ChatGPT. Ultimately, this evaluation not only measures SADA's detection accuracy but also demonstrates its capability to identify hard-to-detect vulnerabilities, which sets it apart from tools like Slither and ChatGPT.

6.3. Qualitative Assessment

In addition to the quantitative analysis, a qualitative assessment is performed by us to evaluate the quality of the reports generated by SADA. This evaluation focuses on the following specific criteria by rating each on a scale of one through ten:

- **Clarity:** The extent to which the report is easy to read, understand, and interpret, with minimal ambiguity.
- **Actionability:** The degree to which the report provides practical, clear, and specific recommendations for addressing identified vulnerabilities.
- **Depth:** The comprehensiveness and technical rigor of the analysis, including detailed explanations and coverage of a wide range of vulnerabilities.

A key component of the qualitative assessment is examining how clearly SADA explains the vulnerabilities it detects. The reports should not only identify the vulnerabilities but also clearly explain the potential risks associated with each vulnerability, as well as recommend specific actions for remediation. For example, the report should describe the nature of the issue, such as a reentrancy attack or access control flaw, and provide detailed suggestions for addressing the problem, such as changing access modifiers or adding checks to prevent unauthorized calls. The report should include code snippets to illustrate how vulnerabilities can be mitigated or resolved to assist smart contract developers appropriately.

The depth of each report is assessed by evaluating the level of detail in the explanations and recommendations. A well-rounded report should address why a particular vulnerability poses a risk, how it can be exploited, and why specific changes are recommended to mitigate it. The quality of the recommendations will be assessed based on their practicality and specificity 釱摿 hether they are actionable in real-world scenarios and whether they align with best practices in smart contract security. Ideally, the report should be written well enough for a novice smart contract developer to understand.

7. Results

This section summarizes the quantitative and qualitative testing performed with the eleven smart contracts to identify vulnerabilities.

7.1. Quantitative Assessment Summary

Precision is the proportion of true positive predictions among all positive predic-

tions made by the model [19]. Precision is calculated by:

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}}$$

The results indicate that SADA's precision ranges from 0.83 to 1.00 for all of the smart contracts tested, with an average precision of 0.93. In contrast, the precision of simply using ChatGPT ranges from 0.67 to 0.89, with an average precision of 0.77. The superior precision of SADA compared to ChatGPT suggests that SADA is more reliable in identifying actual vulnerabilities with fewer false alarms. This difference can be attributed to SADA's ability to perform both static analysis and dynamic analysis, whereas ChatGPT only performs static analysis. Furthermore, SADA incorporates domain-specific knowledge thanks to the use of the fine-tuned GPT-4o model for static analysis.

Recall measures the proportion of actual vulnerabilities that were correctly identified [19], and is calculated as:

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}}$$

A high recall indicates that the model is effective at detecting a large proportion of existing vulnerabilities. In this study, false negatives also include cases where both Slither and SADA, or Slither and ChatGPT, failed to detect vulnerabilities. The results indicate that SADA's recall ranges from 0.81 to 1.00 for all of the smart contracts tested, with an average recall of 0.93. In contrast, the precision of simply using ChatGPT ranges from 0.46 to 1.00, with an average recall of 0.73.

The F1 score is a crucial performance metric in machine learning that provides a balanced assessment of a model's accuracy, especially for classification tasks [20]. It combines two important metrics, precision and recall, into a single value, which offers a more comprehensive evaluation than accuracy alone. The F1 score is the harmonic mean of precision and recall, which is calculated as:

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

Generally, a high F1 score indicates that a machine learning model excels at identifying true positives while minimizing both false positives and false negatives. In this evaluation, SADA's F1 score ranges from 0.85 to 0.97 for all of the smart contracts tested, with an average F1 score of 0.92. In contrast, ChatGPT's F1 score ranges from 0.55 to 0.94, with an average F1 score of 0.75. SADA's consistently high F1 score showcases its ability to maintain a comprehensive approach to detecting vulnerabilities in smart contracts, both statically and dynamically. This balance is critical in smart contract security, where false positives can overwhelm developers with unnecessary remediation efforts, and false negatives can leave critical vulnerabilities undetected. The model's robustness across both small and large smart contracts indicates that its dual-analysis framework is successful in addressing a wide range of code patterns and vulnerabilities.

Table 3 summarizes the quantitative analysis results for SADA, and **Table 4**

does the same for ChatGPT. SADA's superior performance stems from its dual-analysis framework, which integrates static and dynamic methods. For instance, SADA identified additional vulnerabilities not labeled by Slither, such as reentrancy vulnerabilities and ownership-related vulnerabilities. This flexibility demonstrates SADA's capacity to generalize beyond pre-existing datasets.

Table 3. Quantitative results from SADA.

Smart Contract	Vulnerabilities Labeled by Slither	True Positives from SADA	False Positives from SADA	False Negatives from SADA	Vulnerabilities Found by SADA but Missed by Slither	Precision of SADA	Recall of SADA	F1 Score of SADA
Contract 1	10	17	0	4	9	1.00	0.81	0.89
Contract 2	1	13	1	0	12	0.93	1.00	0.96
Contract 3	4	11	1	1	7	0.92	0.92	0.92
Contract 4	2	11	1	0	9	0.92	1.00	0.96
Contract 5	2	8	0	1	7	1.00	0.89	0.94
Contract 6	2	15	1	0	13	0.94	1.00	0.97
Contract 7	7	13	2	2	6	0.87	0.87	0.87
Contract 8	4	13	1	2	10	0.93	0.87	0.90
Contract 9	4	8	0	1	5	1.00	0.89	0.94
Contract 10	0	5	1	0	5	0.83	1.00	0.91
Contract 11	0	8	1	0	8	0.89	1.00	0.94
Averages						0.93	0.93	0.93

Table 4. Quantitative results from ChatGPT (using GPT-4o).

Smart Contract	Vulnerabilities Labeled by Slither	True Positives from ChatGPT	False Positives from ChatGPT	False Negatives from ChatGPT	Precision of ChatGPT	Recall of ChatGPT	F1 Score of ChatGPT
Contract 1	10	7	2	4	0.78	0.64	0.70
Contract 2	1	6	3	7	0.67	0.46	0.55
Contract 3	4	9	4	5	0.69	0.64	0.67
Contract 4	2	10	2	4	0.83	0.71	0.77
Contract 5	2	8	1	0	0.89	1.00	0.94
Contract 6	2	7	3	5	0.70	0.58	0.64
Contract 7	7	9	2	4	0.82	0.69	0.75
Contract 8	4	10	3	3	0.77	0.77	0.77
Contract 9	4	7	3	3	0.70	0.70	0.70
Contract 10	0	5	1	0	0.83	1.00	0.91
Contract 11	0	7	2	1	0.78	0.88	0.82
Averages					0.77	0.73	0.75

7.2. Qualitative Assessment Summary

In addition to the quantitative evaluation of SADA's performance, a qualitative assessment is conducted to evaluate the clarity, actionability, and depth of its generated reports. These attributes are critical for ensuring that the tool not only identifies vulnerabilities accurately but also provides developers with the necessary information to address them effectively. Table 5 summarizes the qualitative assessments of the smart contracts used for validation.

Table 5. Qualitative results.

Smart Contract	Clarity (1 - 10)	Actionability (1 - 10)	Depth (1 - 10)	Total (out of 30)	Limitations or Observations
Contract 1	9	10	9	28	Clear descriptions with severity ratings for each vulnerability. Report provides specific code snippets to fix.
Contract 2	7	7	6	20	Vulnerabilities reported are clear but lack examples tailored to the contract's context.
Contract 3	9	10	10	29	Exceptional detail and highly actionable and specific recommendations.
Contract 4	10	10	10	30	Actionable, clear, and detailed; well-suited for developers of varying experience.
Contract 5	8	7	6	21	Severity rankings are out of order. Some suggestions could be more specific to prevent misinterpretation.
Contract 6	8	8	9	25	Most recommendations are clear and actionable but some could use more detail, e.g. for "gas inefficiency".
Contract 7	7	7	8	22	Dynamic analysis insights lack specific and actionable code fixes. Static analysis is detailed.
Contract 8	8	6	6	20	Dynamic analysis could benefit from more specific code fixes. Report is decently organized.
Contract 9	9	10	10	29	Comprehensive and well-organized by static and dynamic vulnerabilities, with excellent examples and actionable steps.
Contract 10	10	10	10	30	Detailed and highly actionable; excellent depth in explanation.
Contract 11	8	9	8	25	Some recommendations are broad; clarity is good but could benefit from more examples.
Averages	8.45	8.55	8.36	25.36	

The qualitative assessment of the reports generated by SADA for the smart contracts indicates generally strong performance across the three key criteria: clarity, actionability, and depth. The average score across all contracts is 25.36 out of 30.

Contract 4 and Contract 10 stand out with perfect scores (30), providing exceptional clarity, highly actionable recommendations, and thorough depth in their analysis. These reports are well-suited for developers with varying levels of experience. For instance, as shown in Figure 4, Contract 4 clearly points out a potential for overflow by the code's random number generator, and it provides a piece of code that can mitigate this vulnerability.

Synthesis of Static and Dynamic Analysis Findings

Identified Vulnerabilities and Recommendations

7. Potential for Overflows in Random Number Generation

- **Description**: Lack of boundary checks for random indices could lead to overflows.
- **Risk**: Low. Could result in incorrect token assignments.
- **Affected Functions**: `useRandomAvailableToken`.
- **Recommendations**:
  - Ensure indices are within expected bounds.

```
// Ensure random index is within bounds
uint256 randomIndex = randomNum % _numAvailableTokens;
require(randomIndex < _numAvailableTokens, "Random index out of bounds");
```

Figure 4. Snapshot of a vulnerability description in Contract 4.

Similarly, Contracts 3 and 9 also scored highly, with scores of 29 out of 30 each, thanks to comprehensive and well-organized reports that include excellent, specific examples and actionable steps. As shown in Figure 5, Contract 9 provides an exceptional example here, with a race condition risk that it identifies as well as accurate and actionable recommendations on how to fix it.

Synthesis of Static and Dynamic Analysis Findings

Identified Vulnerabilities and Recommendations

6. Approval Race Condition (ERC20 Issue)

- **Description**: The `approve` function is vulnerable to a race condition where spenders can exploit the allowance before it changes.
- **Risk**: Unauthorized transfers if the spender acts quickly.
- **Severity**: High
- **Exploitation**: Spenders can transfer more tokens than intended.
- **Mitigation**:
  - Use `increaseAllowance` and `decreaseAllowance` instead of `approve`.

```
function increaseAllowance(address spender, uint256 addedValue) public returns (bool)
{
    allowed[msg.sender][spender] = allowed[msg.sender][spender].add(addedValue);
    emit Approval(msg.sender, spender, allowed[msg.sender][spender]);
    return true;
}
```

Figure 5. Snapshot of a vulnerability description in Contract 9.

However, some reports show room for improvement. Contracts 2, 5, and 8 scored lower, with total scores ranging from 20 to 22. These reports were noted for lacking specific examples or context-tailored suggestions, which could limit their effectiveness for developers. For instance, Contract 5 had issues with the order and clarity of severity rankings, while Contract 8's dynamic analysis lacked specific code fixes. In Contract 8, some issues related to slippage protection, liquidity risks, and potential exploitation from arithmetic vulnerabilities were found, but there is not enough actionable detail to mitigate these issues.

This qualitative assessment suggests that while SADA excels in providing clear and actionable reports, there is still potential for enhancing the specificity of recommendations and improving the organization of the reports.

8. Analysis and Comparison with Prior Work

Static analysis tools, dynamic analysis tools, and GPT have been widely used for

their ability to identify potential vulnerabilities in Solidity code. However, they each come with distinct strengths and weaknesses when compared to SADA, which combines static and dynamic analysis for more comprehensive and thorough vulnerability detection.

Static analysis tools, such as Oyente, Osiris, and Slither, excel at finding common vulnerabilities that are detectable by code inspection. In one comparative study by Ottati *et al.*, Slither is found to be the "best tool for detecting reentrancy (100%) and unchecked low-level calls (87%), but Oyente had better performance detecting Underflow (33%) and Osiris did better on Overflow (53%)" [21]. In general, static analysis tools are effective at identifying known vulnerabilities before a contract is deployed. However, they may struggle with dynamic issues that appear only during runtime, such as state-dependent vulnerabilities or those triggered by specific contract interactions. They are also limited to identifying common vulnerability structures and struggle with novel vulnerabilities that do not follow typical patterns.

Dynamic analysis tools, such as Mythril and Manticore, examine the behavior of contracts during runtime to identify vulnerabilities and potential issues. This approach complements static analysis by detecting problems that may only manifest during execution. These tools generally operate by symbolic execution (exploring multiple paths a smart contract could take under various inputs), fuzzing (inputting invalid data to monitor a smart contract's response), and concolic analysis (a combination of concrete and symbolic execution, used by tools like Mythril to explore contract behavior under various conditions) [22]. However, dynamic analysis tools are notorious for false positives. In one study by Kumar *et al.*, Mythril had a higher false positive ratio of 12% compared to static tools like Slither (8%) and Oyente (7%) [23]. Dynamic analysis tools like Manticore and Mythril have also been described as having long execution times, taking 5 - 10 minutes for relatively simple contracts and freezing when given a more complex contract [24].

With the advent of OpenAI's GPT LLM agents, many researchers have explored the use of GPT to analyze smart contracts for vulnerabilities. However, their work is mostly limited to static analysis of contracts or combining GPT with static analysis. Recent studies have shown promising results in this direction, but also revealed some limitations. GPTScan, for instance, is a tool that combines GPT with static analysis for detecting logic vulnerabilities in smart contracts [25]. It achieves high precision (over 90%) for token contracts and acceptable precision (57.14%) for large projects [25]. GPTScan breaks down each logic vulnerability type into scenarios and properties, using GPT as a code understanding tool rather than relying solely on it for vulnerability detection [25]. Another study evaluated ChatGPT's effectiveness in identifying smart contract vulnerabilities using the SmartBugs dataset [26]. The results showed that while ChatGPT achieves high recall rates (88.2% for GPT-4), its precision in pinpointing vulnerabilities is relatively low (22.6% for GPT-4) [26]. Performance varies across different types of vulnerabilities, with ChatGPT performing well in detecting certain issues like unchecked re-

turn values [26].

SADA represents a significant advancement in smart contract vulnerability detection, and it offers several key advantages over existing tools. Since it combines both static and dynamic analysis techniques, SADA achieves an impressive balance of precision (0.93) and recall (0.92), outperforming many current solutions described above. SADA intends to address many of the shortcomings of the existing tools described above, such as high false positive rates, lack of comprehensive vulnerability detection, and long execution times.

Table 6 provides a comparison of various state-of-the-art tools against SADA.

Table 6. Comparison of various tools for smart contract vulnerability detection.

Tool Type	Examples	Strengths	Weaknesses
Static Analysis	Oyente, Osiris, Slither	Detect common vulnerabilities; effective before deployment	Limited to static patterns; struggles with dynamic or novel issues
Dynamic Analysis	Mythril, Manticore	Detect runtime vulnerabilities; complements static analysis	High false positives; long execution times; freezes on complex contracts
GPT-Based Analysis	GPTScan, ChatGPT	High precision for certain vulnerability types; scalable for logic vulnerabilities	Low precision for pinpointing vulnerabilities; limited recall for novel issues
SADA (Proposed Tool)	N/A	High precision (0.93) and recall (0.92); balances static and dynamic analysis	Computationally expensive

9. Challenges and Limitations

A tool like SADA presents inherent limitations and challenges. SADA is currently designed to work with completed and deployed smart contracts, so additional functionality is needed to audit smart contracts that are still in the development stages. However, SADA in its current form can be used, for example, to detect whether a token's smart contract is malicious, serving a purpose similar to tools like TokenSniffer.

However, SADA's reliance on an LLM introduces challenges. The potential for hallucinations—where the LLM generates convincing but incorrect information—and false positives remain a concern. This can lead to misinterpretation of the results, especially if SADA is used without human oversight. Also, the evolving nature of blockchain technology means new types of vulnerabilities may emerge, which SADA may be unable to detect since it is limited to its pre-trained knowledge database. Retrieval-augmented generation (RAG) may prove to be useful here. Also, SADA is also limited to Ethereum-based contracts developed in the Solidity programming language. SADA also may struggle with long or complex smart contracts, or smart contracts containing multiple files, due to its limited context window. SADA should be used in conjunction with a human smart contract auditor to validate the generated report and ensure its accuracy.

10. Conclusions and Future Work

The development and evaluation of SADA for smart contract vulnerability detec-

tion represents a significant step forward in enhancing Ethereum blockchain security. This research has demonstrated that combining static and dynamic analysis techniques with LLM agents can yield superior results compared to traditional approaches. Key findings from SADA include improved accuracy, comprehensive coverage, reduced false positives, and adaptability to various styles of Solidity smart contracts compared to current tools.

SADA offers significant potential for improvement and future development. A key area for enhancement lies in achieving more comprehensive vulnerability detection to reduce its false positive rate. For example, SADA could be trained on an even broader range of vulnerabilities or utilize a custom LLM designed specifically for smart contract security. Expanding SADA's capabilities to support multiple blockchain platforms, such as Solana and Polkadot, would further increase its utility and accessibility for developers across different ecosystems.

Acknowledgements

We thank the reviewers for their detailed comments that greatly improved the paper.

Conflicts of Interest

The authors declare no conflicts of interest regarding the publication of this paper.

References

[1] IBM (2024) What Are Smart Contracts on Blockchain?

[2] SWC Registry (2024) Smart Contract Weakness Classification (SWC) Registry, 2020.

[3] Wikipedia Contributors (2024) The DAO—Wikipedia.

[4] Luu, L., Chu, D., Olickel, H., Saxena, P. and Hobor, A. (2016) Making Smart Contracts Smarter. *Proceedings of the* 2016 *ACM SIGSAC Conference on Computer and Communications Security*, Vienna, 24-28 October 2016, 254-269. https://doi.org/10.1145/2976749.2978309

[5] Consensys Diligence (2019) What Is Mythril? Mythril v 0.23.9 Documentation.

[6] Mossberg, M., Manzano, F., Hennenfent, E., Groce, A., Grieco, G., Feist, J., *et al.* (2019) Manticore: A User-Friendly Symbolic Execution Framework for Binaries and Smart Contracts. 2019 34*th IEEE*/*ACM International Conference on Automated Software Engineering* (*ASE*), San Diego, 11-15 November 2019, 1186-1189. https://doi.org/10.1109/ase.2019.00133

[7] Feist, J., Grieco, G. and Groce, A. (2019) Slither: A Static Analysis Framework for Smart Contracts. 2019 *IEEE*/*ACM* 2*nd International Workshop on Emerging Trends in Software Engineering for Blockchain* (*WETSEB*), Montreal, 27 May 2019, 8-15. https://doi.org/10.1109/wetseb.2019.00008

[8] Boi, B., Esposito, C. and Lee, S. (2024) Smart Contract Vulnerability Detection: The Role of Large Language Model (LLM). *ACM SIGAPP Applied Computing Review*, **24**, 19-29. https://doi.org/10.1145/3687251.3687253

[9] He, Z., Zhao, Z., Chen, K. and Liu, Y. (2024) Smart Contract Vulnerability Detection Method Based on Feature Graph and Multiple Attention Mechanisms. *Computers, Materials & Continua*, **79**, 3023-3045. https://doi.org/10.32604/cmc.2024.050281

[10] Ma, W., Wu, D., Sun, Y., Wang, T., Liu, S., Zhang, J., Xue, Y. and Liu, Y. (2024) Combining Finetuning and LLM-Based Agents for Intuitive Smart Contract Auditing with Justifications. https://arxiv.org/abs/2403.16073

[11] BasuMallick, C. (2023) Smart Contracts: Types, Benefits, and Tools. Spiceworks.

[12] OWASP Foundation (2024) OWASP Smart Contract Top 10.

[13] Trust Wallet (2024) What Is a Mempool in Crypto?

[14] Rossini, M. (2022) Slither Audited Smart Contracts Dataset. Hugging Face.

[15] Datadog (2024) What Is Static Analysis?

[16] Ethereum Foundation (2024) Web3.py: A Python Interface for Interacting with the Ethereum Blockchain and Ecosystem.

[17] Alchemy (2022) What Is an ABI of a Smart Contract?

[18] Wikipedia Contributors (2024) Precision and Recall.

[19] Tigerschiold, T. (2022) What Is Accuracy, Precision, Recall and F1 Score? Labelf Blog.

[20] Ibrahim, M. (2024) An Introduction to the F1 Score in Machine Learning. Weights & Biases.

[21] Ottati, J., Ibba, G. and Rocha, H. (2023) Comparing Smart Contract Vulnerability Detection Tools. *The 22nd Belgium-Netherlands Software Evolution Workshop*, Nijmegen, 27-28 November 2023, 1-16.

[22] Lashkari, B. and Musilek, P. (2023) Evaluation of Smart Contract Vulnerability Analysis Tools: A Domain-Specific Perspective. *Information*, **14**, Article 533. https://doi.org/10.3390/info14100533

[23] Sharath Kumar, D.R.V.A., Mishra, A., Muthupandi, G., Sivavara Prasad, J. and Upadhyay, T. (2024) An In-Depth Analysis and Performance of Existing Techniques for Ethereum Smart Contract Vulnerability Detection. *Journal of Electrical Systems*, **20**, 8294-8301.

[24] Dmitrikov, D. and Piqueras, E. (2020) The Landscape of Solidity Smart Contract Security Tools in 2020. Kleros.

[25] Sun, Y., Wu, D., Xue, Y., Liu, H., Wang, H., Xu, Z., *et al.* (2024) GPTScan: Detecting Logic Vulnerabilities in Smart Contracts by Combining GPT with Program Analysis. *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, Lisbon, 14-20 April 2024, 1-13. https://doi.org/10.1145/3597503.3639117

[26] Chen, C., Su, J., Chen, J., Wang, Y., Bi, T., Yu, J., *et al.* (2024) When ChatGPT Meets Smart Contract Vulnerability Detection: How Far Are We? *ACM Transactions on Software Engineering and Methodology*. https://doi.org/10.1145/3702973

11 Appendices

11.1 System Prompt for Static Analysis Agent

Below is the system prompt used to initiate the model:

Analyze the provided Solidity smart contract code for vulnerabilities and suggest improvements. Identify the specified vulnerabilities, indicate their locations, and provide detailed recommendations for fixes. Addition-ally, highlight any other vulnerabilities you find and propose code optimizations.

Steps

1. **Code Review**: Thoroughly examine the provided Solidity smart contract code.
2. **Identify Known Issues**: Locate the specified vulnerabilities within the code.
3. **Highlight Other Vulnerabilities**: Identify any additional vulnerabilities with a high level of confidence.
4. **Recommendations for Fixes**: Provide detailed, actionable solutions for addressing each vulnerability.
5. **Code Optimization**: Suggest enhancements to improve the overall efficiency and performance of the code. Output Format The output should be a structured analysis in a para-graph format, detailing:

- Vulnerability location: Indicate the specific line or section of the code where each issue is found.
- Recommended fixes: Provide clear instructions on how to resolve each identified vulnerability.
- Additional vulnerabilities: Mention any other vulnerabilities observed, along with their locations and solutions.
- Code optimizations: Suggest potential code improvements for efficiency and performance. ** Examples ** **Input** Solidity Smart Contract Code:

```
// Provided Solidity contract code
pragma solidity ^0.8.0;
contract Example {
    mapping(address => uint256) private balances;
    function withdraw(uint256 amount) public {
        if (balances[msg.sender] >= amount) {
            (bool success, ) = msg.sender.call{value: amount}("");
            require(success, "Transfer failed.");
            balances[msg.sender] -= amount;
        }
    }
}
```

Output

- **Reentrancy Vulnerability**: Located in the `withdraw` function. Fix by updating the state before making external calls:
  ```
  require(balances[msg.sender] >= amount);
  balances[msg.sender] -= amount;
  (bool success, ) = msg.sender.call{value: amount}("");
  require(success, "Transfer failed.");
  ```
- **Use of** `call`: Consider replacing `call` with `send` or `transfer` if the gas limit constraint is man-ageable.
- **Code Optimization**: Use `safeMath` library to handle arithmetic operations securely in older Solidity versions. **Notes**
- Ensure the Solidity version is compatible with suggested code optimizations and vulnerability fixes. Keep up-to-date with the latest security best practices and large-scale vulnerabilities affecting Solidity.

11.2 Static Analysis Report Example

Static Response

Analysis of the `DMC` Smart Contract

1. Ownership and Control

- **Issue:** The contract uses a custom ownership mechanism with the `newComer` variable instead of the standard `Ownable` pattern. The `onlyOwner` modifier checks against `newComer` rather than `_owner`.

- **Impact:** This could lead to confusion or unintended behavior since the ownership is not managed in the usual way. Additionally, the `transferOwnership` function is private, meaning ownership cannot be transferred publicly.
- **Recommendation:** Use the standard `Ownable` pattern to manage ownership. Make the `transferOwnership` function public or external to allow ownership transfer.

2. Approval Race Condition (SWC-114)

- **Issue:** The contract does not implement the "safe approval" pattern, which can lead to race conditions when changing allowances. This is a known issue in ERC20 tokens.
- **Impact:** Medium. If a spender is quick enough, they could use the allowance before it is changed, leading to unintended transfers.
- **Recommendation:** Use the `increaseAllowance` and `decreaseAllowance` functions instead of directly modifying the allowance. Alternatively, you can implement the `safeApprove` pattern:

```
function approve(address spender, uint256 value) public returns (bool) {
    require(spender != address(0));
    require(_allowances[msg.sender][spender] == 0 || value == 0, "ERC20: Non-zero allowance must
    _allowances[msg.sender][spender] = value;
    emit Approval(msg.sender, spender, value);
    return true;
}
```

3. Locked Ether (SWC-100)

- **Issue:** The contract does not have a mechanism to withdraw Ether that might be accidentally sent to it.
- **Impact:** Low. Since the contract is an ERC20 token, it is not supposed to handle Ether. However, if Ether is accidentally sent to the contract, it will be locked there permanently.
- **Recommendation:** Implement a `withdrawEther` function that allows the owner to recover any Ether accidentally sent to the contract. For example:

```
function withdrawEther() external onlyOwner {
    address payable ownerAddress = address(uint160(owner()));
    ownerAddress.transfer(address(this).balance);
}
```

4. Incorrect Equality Check in `_approve`

- **Issue:** The `_approve` function has a conditional check that resets the allowance to zero if the `ol` (owner) is not the contract owner:

```
if (ol != owner()) { _allowances[ol][tt] = 0; emit Approval(ol, tt, 4); }
else { _allowances[ol][tt] = amount; emit Approval(ol, tt, amount); }
```

  This behavior is unusual and can lead to unexpected allowance resets.
- **Impact:** Medium. This can cause issues for users who are not the contract owner, as their allowances will be reset to zero unexpectedly.
- **Recommendation:** Clarify the purpose of this check and ensure that it aligns with the intended behavior. If the goal is to prevent non-owners from setting allowances, consider adding a clear comment or modifying the logic to make it more intuitive.

5. Gas Inefficiency in `SafeMath`

- **Issue:** The `SafeMath` library includes checks that are unnecessary in certain cases. For example, the `mul` function checks if `a` is zero before performing the multiplication:

```
function mul(uint256 a, uint256 b) internal pure returns (uint256) {
    if (a == 0) {
        return 0;
    }
```

```
        uint256 c = a * b;
        require(c / a == b, "SafeMath: multiplication overflow");
        return c;
    }
```

This check is unnecessary in most cases and can be removed to save gas.

- **Impact:** Low. This only affects gas costs.
- **Recommendation:** Consider removing the unnecessary checks in `SafeMath` to optimize gas usage.

6. Reentrancy Risk in `sendValue` (SWC-107)

- **Issue:** The `sendValue` function uses a low-level call to transfer Ether, which can trigger a reentrancy attack if the recipient is a contract.
- **Impact:** Medium. Although the contract is not designed to handle Ether, if Ether is accidentally sent to the contract and then transferred using `sendValue`, it could trigger a reentrancy attack.
- **Recommendation:** Use the "Checks-Effects-Interactions" pattern to prevent reentrancy. Additionally, consider using `ReentrancyGuard` to add reentrancy protection.

Conclusion

The `DMC` contract has several issues related to ownership management, allowance handling, and potential reentrancy risks. By addressing these issues and following best practices, the contract can be made more secure. Specifically, the ownership mechanism should be clarified, and the allowance handling should be improved to prevent race conditions. Additionally, adding a mechanism to recover locked Ether and protecting against reentrancy in `sendValue` would enhance the contract's security.

11.3 Dynamic Analysis Report Example

Dynamic Response

Analysis of the Smart Contract

1. Swap-Related Vulnerabilities

- **Slippage Protection**:
  - **Issue**: The contract does not explicitly show any mechanism for slippage protection. This could lead to unintended losses during trades if the price changes significantly between the time a transaction is initiated and when it is executed.
  - **Recommendation**: Implement slippage protection by allowing users to specify a minimum acceptable amount when executing a swap. For example:

```
function swap(uint256 amountIn, uint256 minAmountOut) external {
    uint256 amountOut = getSwapAmount(amountIn);
    require(amountOut >= minAmountOut, "Slippage exceeded");
    // Proceed with the swap
}
```

- **Front-Running and Miner Manipulation**:
  - **Issue**: There is no evidence of mechanisms to prevent front-running, such as time-locks or commit-reveal schemes. This makes the contract vulnerable to front-running attacks, where miners or other users can manipulate the order of transactions to their advantage.
  - **Recommendation**: Implement a mechanism like a commit-reveal scheme or a delay in the execution of swaps to mitigate front-running. For example:

```
function commitSwap(uint256 amountIn) external {
    // Commit the swap without revealing the details
}

function revealSwap(uint256 amountIn, uint256 minAmountOut) external {
    // Reveal the swap details and execute it
}
```

- **Liquidity Risks**:
  - **Issue**: The contract does not explicitly handle low liquidity scenarios. This could lead to failed transactions or excessive slippage if there is not enough liquidity in the pool.
  - **Recommendation**: Add checks to ensure that there is sufficient liquidity before executing a swap. For example:

```
function swap(uint256 amountIn) external {
    require(liquidityPool >= amountIn, "Insufficient liquidity");
    // Proceed with the swap
}
```

- **Re-Entrancy Risks**:
  - **Issue**: The contract does not appear to have any re-entrancy protections in place. This could be a risk if external calls are made during the swap process.
  - **Recommendation**: Use the "Checks-Effects-Interactions" pattern to prevent re-entrancy. Additionally, consider using `ReentrancyGuard` to add re-entrancy protection. For example:

```
function swap(uint256 amountIn) external nonReentrant {
    // Perform checks
    // Update state
    // Interact with external contracts
}
```

2. Access Control Issues

- **Permissioned Access**:
  - **Issue**: The contract includes functions like `renounceOwnership` and `approve` that are restricted to the owner. However, the error messages indicate that these functions are being called by unauthorized users:
    * `execution reverted: Ownable: caller is not the owner`
    * `execution reverted: ERC20: approve from the zero address`
  - **Recommendation**: Ensure that only the contract owner or authorized users can call sensitive functions. For example:

```
modifier onlyOwner() {
    require(msg.sender == owner, "Caller is not the owner");
    _;
}

function renounceOwnership() external onlyOwner {
    // Renounce ownership
}
```

- **Role Escalation**:
  - **Issue**: There is no evidence of role escalation vulnerabilities in the contract. However, it's important to ensure that only authorized users can perform actions that could affect the contract's state.
  - **Recommendation**: Implement role-based access control to prevent unauthorized users from gaining elevated privileges. For example:

```
mapping(address => bool) private admins;

modifier onlyAdmin() {
    require(admins[msg.sender], "Caller is not an admin");
    _;
}

function addAdmin(address _admin) external onlyOwner {
    admins[_admin] = true;
}
```

- **Ownership and Control Transfer**:

- **Issue**: The `renounceOwnership` function is restricted to the owner, but the error message indicates that it was called by a non-owner:
  - `execution reverted: Ownable: caller is not the owner`
- **Recommendation**: Ensure that ownership transfer functions are only callable by the current owner. Additionally, consider adding a mechanism to transfer ownership securely, such as a two-step process:

```
address private pendingOwner;

function transferOwnership(address _newOwner) external onlyOwner {
    pendingOwner = _newOwner;
}

function acceptOwnership() external {
    require(msg.sender == pendingOwner, "Caller is not the pending owner");
    owner = pendingOwner;
    pendingOwner = address(0);
}
```
```\b

**3.** Potential Exploits

- **Arithmetic Vulnerabilities**:
  - **Issue**: The contract does not use `SafeMath` for arithmetic operations, which could lead to overflow or underflow vulnerabilities, especially in functions like `transfer` and `approve`.
  - **Recommendation**: Use `SafeMath` for all arithmetic operations to prevent overflows and underflows. For example:

```
using SafeMath for uint256;

function transfer(address _to, uint256 _value) external {
    balances[msg.sender] = balances[msg.sender].sub(_value);
    balances[_to] = balances[_to].add(_value);
    emit Transfer(msg.sender, _to, _value);
}
```

- **Denial of Service (DoS) Vectors**:
  - **Issue**: The contract does not have any obvious loops or storage-heavy operations that could be exploited for DoS attacks. However, functions like `transfer` and `approve` could be abused if large amounts of data are processed.
  - **Recommendation**: Limit the amount of data processed in each transaction and add checks to prevent excessive gas consumption. For example:

```
function transfer(address _to, uint256 _value) external {
    require(_value <= maxTransferAmount, "Transfer amount exceeds limit");
    // Proceed with the transfer
}
```

- **Unauthorized Token Transfers**:
  - **Issue**: The contract includes errors like `BEP20: transfer from the zero address` and `BEP20: transfer amount exceeds balance`, indicating that unauthorized or incorrect transfers are being attempted.
  - **Recommendation**: Add checks to ensure that transfers are only made from valid addresses and that the transfer amounts are within the allowed limits. For example:

```
function transfer(address _to, uint256 _value) external {
    require(msg.sender != address(0), "Transfer from zero address");
    require(_to != address(0), "Transfer to zero address");
    require(balances[msg.sender] >= _value, "Insufficient balance");
    // Proceed with the transfer
}
```
```

4. Security Recommendations

- **Slippage Protection**: Implement slippage protection in swap functions to prevent unintended losses due to price changes.
- **Front-Running Mitigation**: Use mechanisms like commit-reveal or time-locks to prevent front-running attacks.
- **Re-Entrancy Protection**: Implement the "Checks-Effects-Interactions" pattern and consider using `ReentrancyGuard` to prevent re-entrancy attacks.
- **Access Control**: Ensure that sensitive functions are restricted to authorized users, and implement role-based access control to prevent unauthorized access.
- **SafeMath**: Use `SafeMath` for all arithmetic operations to prevent overflows and underflows.
- **Ownership Transfer**: Implement a secure ownership transfer process, such as a two-step transfer, to prevent unauthorized ownership changes.
- **Error Handling**: Improve error handling in functions like `transfer` and `approve` to prevent unauthorized or incorrect transfers.

5. Execution Reverted: Ownable: Caller is Not the Owner

- **Issue**: The error "execution reverted: Ownable: caller is not the owner" indicates that a function restricted to the contract owner was called by an unauthorized user. This is important because it shows that the contract correctly enforces access control, preventing unauthorized users from executing sensitive functions.
- **Recommendation**: Ensure that all owner-only functions are properly protected with the `onlyOwner` modifier, and educate users about the importance of calling these functions only as the contract owner.

Conclusion

The contract has several vulnerabilities related to slippage protection, front-running, and access control. By implementing the recommended changes, such as slippage protection, front-running mitigation, and reentrancy protection, the contract can be made more secure. Additionally, using `SafeMath` and improving error handling will help prevent common exploits like overflows and unauthorized transfers.

11.4 Synthesized Report Example

Synthesized Response

Synthesis of Static and Dynamic Analysis Findings

1. Ownership and Control Issues

- **Issue Description:** The contract utilizes a custom ownership mechanism with a `newComer` variable, deviating from the standard `Ownable` pattern. The `transferOwnership` function is private, restricting ownership changes to internal calls.
- **Security Risk:** This setup can lead to misunderstandings or mismanagement of ownership, potentially resulting in unauthorized access or denial of service.
- **Affected Functions:** Primarily the ownership-related functions, notably the `transferOwnership`.
- **Severity Level:** Moderate.
- **Exploitation Methods:** An attacker might exploit the lack of clarity to assume control or deny rightful ownership changes.
- **Recommendations:**
  - Implement the standard `Ownable` pattern from OpenZeppelin to manage ownership.
  - Make the `transferOwnership` function external to allow for public ownership transfers.
  - Code Snippet:

```
contract DMC is Ownable {
    // Use standard Ownable from OpenZeppelin
}
```

2. Approval Race Condition (SWC-114)

- **Issue Description:** The contract lacks a safe approval mechanism, which can lead to race conditions with allowance changes.
- **Security Risk:** Medium. A malicious spender could exploit this condition to spend more tokens than intended if they act quickly.
- **Affected Functions:** The `approve` function.
- **Severity Level:** Medium.
- **Exploitation Methods:** Front-runners can use this race condition to conduct unauthorized transactions.
- **Recommendations:**
  - Implement `increaseAllowance` and `decreaseAllowance` instead of directly modifying allowances.
  - Use the `safeApprove` pattern to ensure allowances are reset to zero before setting a new value.
  - Code Snippet:

```
function approve(address spender, uint256 value) public returns (bool) {
    require(spender != address(0));
    require(_allowances[msg.sender][spender] == 0 || value == 0, "ERC20: Non-zero allowance m
    _allowances[msg.sender][spender] = value;
    emit Approval(msg.sender, spender, value);
    return true;
}
```

3. Locked Ether (SWC-100)

- **Issue Description:** Ether sent accidentally to the contract is permanently locked as there is no withdrawal mechanism.
- **Security Risk:** Low. Although not critical, funds sent accidentally cannot be recovered.
- **Affected Functions:** None directly; this is a structural issue.
- **Severity Level:** Low.
- **Recommendations:**
  - Implement a `withdrawEther` function to allow the owner to recover Ether.
  - Code Snippet:

```
function withdrawEther() external onlyOwner {
    address(owner()).transfer(address(this).balance);
}
```

4. Incorrect Equality Check in `_approve`

- **Issue Description:** The `_approve` function has an unusual check that resets allowances to zero if the caller is not the contract owner.
- **Security Risk:** Medium. This can cause unexpected behavior for users who are not the owner.
- **Affected Functions:** `_approve`.
- **Severity Level:** Medium.
- **Exploitation Methods:** This might not be directly exploitable, but it could disrupt expected behavior for users.
- **Recommendations:**
  - Clarify and revise the logic to ensure it aligns with the intended functionality.
  - Consider removing the check or documenting its necessity clearly.

5. Gas Inefficiency in SafeMath

- **Issue Description:** The `SafeMath` library includes unnecessary checks, such as checking if `a` is zero before multiplication.
- **Security Risk:** Low. This affects gas efficiency but not security.
- **Affected Functions:** SafeMath operations.
- **Severity Level:** Low.
- **Recommendations:**
  - Optimize `SafeMath` functions to remove unnecessary checks for better gas efficiency.

6. Reentrancy Risk in `sendValue` (SWC-107)

- **Issue Description:** The `sendValue` function uses a low-level call that can be exploited for reentrancy attacks.
- **Security Risk:** Medium. Although not designed to handle Ether, if Ether is sent, this could be exploited.
- **Affected Functions:** `sendValue`.
- **Severity Level:** Medium.
- **Exploitation Methods:** An attacker could re-enter the function during an Ether transfer.
- **Recommendations:**
  - Implement the Checks-Effects-Interactions pattern.
  - Consider using `ReentrancyGuard`.
  - Code Snippet:

```
function sendValue(address payable recipient, uint256 amount) external nonReentrant {
    require(address(this).balance >= amount, "Insufficient balance");
    (bool success, ) = recipient.call{value: amount}("");
    require(success, "Transfer failed");
}
```

Additional Dynamic Analysis Findings

1. Swap-Related Vulnerabilities

- **Slippage Protection:** Implement parameters in swap functions to handle slippage and avoid losses.
- **Front-Running Protection:** Use commit-reveal schemes to prevent front-running.
- **Liquidity Checks:** Ensure sufficient liquidity exists before swaps.

2. Access Control Issues

- **Unauthorized Access:** Ensure functions are protected with `onlyOwner` and additional role-based checks where necessary.

3. Potential Exploits

- **Arithmetic Vulnerabilities:** Use `SafeMath` to prevent overflows.
- **Denial of Service:** Limit data processed in transactions to prevent excessive gas consumption.
- **Unauthorized Token Transfers:** Add checks to prevent transfers from or to zero addresses.

Overall Security Posture

While the `DMC` contract contains several vulnerabilities, these can be mitigated with targeted improvements. The primary issues involve ownership management, allowance handling, and potential reentrancy, which can be addressed using standard Solidity patterns and libraries such as `Ownable`, `SafeMath`, and `ReentrancyGuard`. The contract's modularity and adherence to best practices in access control and error handling will significantly enhance its security and resilience against common exploits.