Sligpt: A Large Language Model-Based Approach for Data Dependency Analysis on Solidity Smart Contracts

Xiaolei Ren 1,*,† and Qiping Wei 2,*,†

1 School of Computer Science and Engineering, Macau University of Science and Technology, Avenida Wai Long, Taipa, Macau 999078, China
2 Department of Computer Science and Engineering, The University of Texas at Arlington, 500 UTA Blvd., Arlington, TX 76010, USA
* Correspondence: xren@must.edu.mo (X.R.); qiping.wei@mavs.uta.edu (Q.W.)
† These authors contributed equally to this work.

Abstract: The advent of blockchain technology has revolutionized various sectors by providing transparency, immutability, and automation. Central to this revolution are smart contracts, which facilitate trustless and automated transactions across diverse domains. However, the proliferation of smart contracts has exposed significant security vulnerabilities, necessitating advanced analysis techniques. Data dependency analysis is a critical program analysis method used to enhance the testing and security of smart contracts. This paper introduces Sligpt, an innovative methodology that integrates a large language model (LLM), specifically GPT-4o, with the static analysis tool Slither, to perform data dependency analyses on Solidity smart contracts. Our approach leverages both the advanced code comprehension capabilities of GPT-4o and the advantages of a traditional analysis tool. We empirically evaluate Sligpt using a curated dataset of Ethereum smart contracts. Sligpt achieves significant improvements in precision, recall, and overall analysis depth compared with Slither and GPT-4o, providing a robust solution for data dependency analysis. This paper also discusses the challenges encountered, such as the computational resource requirements and the inherent variability in LLM outputs, while proposing future research directions to further enhance the methodology. Sligpt represents a significant advancement in the field of static analysis on smart contracts, offering a practical framework for integrating LLMs with static analysis tools.

Keywords: Solidity; Ethereum smart contracts; program analysis; data dependencies; GPT-4o; prompt engineering; code understanding; code comprehension

1. Introduction

The emergence of blockchain technology has initiated a significant transformation across various sectors, offering unmatched benefits such as transparency, immutability, and automation [1]. Blockchain technology, which relies on a distributed ledger maintained by a network of nodes, ensures data integrity and security through cryptographic techniques. This decentralized approach eliminates the need for intermediaries, reducing costs and increasing efficiency.

Smart contracts, a key innovation of blockchain technology, are self-executing agreements with the terms of the contract directly written into code. These contracts automatically enforce and execute the agreed-upon terms when predetermined conditions are met, enabling trustless transactions without intermediaries. Platforms like Ethereum have popularized smart contracts, facilitating their adoption in diverse sectors including finance, supply chain management, healthcare, and governance [2–5].

In the finance sector, smart contracts enable the creation of decentralized financial (DeFi) applications, such as lending platforms, decentralized exchanges, and stablecoins. For instance, platforms like Uniswap and Compound leverage smart contracts to automate trading and lending, providing users with greater control over their assets [6]. In supply
chain management, companies like IBM and Maersk use blockchain-based smart contracts to enhance transparency and traceability, ensuring the authenticity of products and reducing fraud [7]. In healthcare, smart contracts can be used to manage patient data securely, facilitate research collaboration, and streamline insurance claims processing [8].

Despite their transformative potential, smart contracts are not without flaws. Significant security vulnerabilities have been exposed, leading to high-profile exploits such as the DAO attack, where an attacker exploited a re-entrancy vulnerability to drain 50 million USD worth of Ether from the decentralized autonomous organization (DAO) [9–11]. Common vulnerabilities include re-entrancy, integer overflow, and underflow, unchecked return values, and access control issues [12,13]. Addressing these security issues is crucial for the widespread adoption and success of smart contracts.

Static analysis has become a cornerstone technique for identifying vulnerabilities within smart contracts, offering comprehensive code coverage and early detection of security flaws without the need for code execution [14]. Data dependency analysis is a major static analysis technique applied to enhance the testing and security analysis of smart contracts. For instance, ILF [15] and SmartTest [16] include the reads and writes of state variables to train models for fuzzing and symbolic execution, respectively. Smartian [17] employs read-and-write data to enhance the effectiveness of fuzzing for smart contracts. SmartExecutor [18] requires dependency data for guided symbolic execution to increase code coverage. Sailfish [19] needs the reads and writes of state variables to detect state inconsistency bugs.

Slither is one of the most widely used static analysis tools in both industry and academia [20]. However, Slither is not without its limitations; it often misses critical state-variable dependencies, resulting in incomplete vulnerability detection [21]. This limitation is particularly problematic, given the diverse and complex nature of smart contract code.

Recent advancements in large language models (LLMs), epitomized by GPT-4o, have demonstrated remarkable capabilities in code comprehension and analysis [22–24]. These models excel in understanding and contextualizing source code, making them ideal for addressing the limitations of traditional static analysis tools.

We propose a novel methodology to harness the code-understanding capability of GPT-4o and the advantages of Slither to perform dependency analyses. The dependency analysis is converted into the process of refining the given dependency data. Slither is utilized to collect the dependency data, and GPT-4o is responsible for refining the data. The refining process is performed by the three roles made from GPT-4o: checker, evaluator, and verifier. The checker filters out the data that do not need to be refined. The evaluator and the verifier form the evaluation-verification loop; they are responsible for refining the to-be-refined data in loops until the verifier accepts the data or the maximum loop limit is reached.

Our approach is novel in two aspects: (a) the dependency analysis is designed as the process of refining the given dependency data. The given data can be easily obtained using a static analysis tool, allowing our approach to take advantage of existing static analysis tools. GPT-4o excels in the refining process as a reasoning tool, effectively leveraging the strengths of both an LLM and a traditional static analysis tool. (b) Our approach introduces three different roles based on GPT-4o to refine the given data. We empirically evaluate our approach using a curated dataset of Ethereum smart contracts, showcasing significant improvements in detection precision, recall, and overall analysis depth. Our findings underscore the potential of combining LLMs with static analysis tools, paving the way for more secure blockchain applications.

The contributions of this study can be encapsulated as follows:

Methodology Proposal: We introduce a comprehensive methodology named Sligpt, which integrates GPT-4o with the Slither static analysis framework to perform data dependency analyses.
Empirical Evaluation: Sligpt is rigorously evaluated using a meticulously curated dataset of Ethereum smart contracts. The results demonstrate enhanced performance in comparison to both Slither and GPT-4o.

Open Source Contribution: We have developed and released Sligpt (The repository can be accessed at https://github.com/contractAnalysis/sligpt, accessed on 30 July 2024), making both the source code and experimental data publicly available for further research and validation.

The structure of this paper is organized as follows: Section 3 delves into the foundational concepts of blockchain, smart contracts, static analysis, and large language models. It provides a comprehensive review of related work in the areas of smart contract security, static analysis, and the application of large language models in code comprehension. Section 4 elaborates on the issues and existing solutions in four specific scenarios and outlines our motivation for this study. Section 5 describes the proposed approach for Sligpt. Section 6 presents the experimental setup and empirical findings of our study. Section 7 and Section 2 discuss the implications of our findings in the context of previous related work. Finally, Section 8 concludes this paper.

2. Related Work

The development and utilization of large language models (LLMs) in software engineering and security analysis have garnered significant attention in recent years. This section reviews related work, categorizing it into three primary areas: code syntax and semantics comprehension, automated program repair, and smart contract security analysis.

2.1. Code Syntax and Semantics Comprehension

Several studies have explored the capabilities of LLMs in understanding and generating code, which is critical for various software engineering tasks. Ma et al. [25] provide an in-depth evaluation of ChatGPT’s abilities to comprehend code syntax and semantic structures, such as abstract syntax trees (AST), control flow graphs (CFG), and call graphs (CG). The study reveals that while ChatGPT performs well in understanding code syntax, it struggles with dynamic semantics, often leading to hallucinations. These findings suggest that further refinement is needed to enhance the interpretability and reliability of LLMs in software engineering applications. This research reinforces the earlier conclusions, highlighting the strengths and limitations of LLMs in static and dynamic code analysis and providing valuable insights for future improvements.

2.2. Automated Program Repair

The utilization of LLMs for automated program repair (APR) has shown promising results. The paper by Huang et al. [26] examines the effectiveness of fine-tuning various LLMs (CodeBERT, GraphCodeBERT, PLBART, CodeT5, UniXcoder) for repairing bugs and vulnerabilities in different programming languages. The study demonstrates that fine-tuned LLMs significantly outperform previous APR tools, offering new strategies to enhance repair capabilities. The research also identifies several limitations and provides guidelines for future work to further leverage LLMs in APR tasks.

2.3. Smart Contract Security Analysis

Smart contract security is a critical area of research due to the increasing prevalence of blockchain technology. The paper by Sun et al. [27] presents GPTScan, a tool that integrates GPT with static analysis for detecting logic vulnerabilities in smart contracts. GPTScan leverages GPT’s code comprehension capabilities to identify potential vulnerabilities, which are then validated through static confirmation. The evaluation results show high precision and recall rates, demonstrating the tool’s effectiveness in detecting new and existing vulnerabilities.

Moreover, Zhang et al. [28] introduces ACFIX, which utilizes GPT-4 and a novel approach to repair access control vulnerabilities in smart contracts. By mining common role-
based access control (RBAC) practices and guiding the LLM with contextual information, ACFIX achieves a high success rate in repairing vulnerabilities, significantly outperforming baseline models.

In summary, previous related work has highlighted substantial progress in utilizing LLMs for software engineering and security analysis tasks. The integration of LLMs with traditional analysis tools, as demonstrated in our work, represents a significant advancement in the field, providing practical solutions and opening new research opportunities.

In our work, we leverage the strengths of GPT-4o and Slither by refining dependency data through three distinct roles: the checker, evaluator, and verifier. This allows our method to capitalize on existing static analysis tools while benefiting from GPT-4o’s reasoning capabilities. Our approach introduces these three distinct roles to effectively combine the advantages of both an LLM and a traditional static analysis tool, thereby improving precision, recall, and overall analysis depth in smart contract security.

3. Background

The rise of blockchain technology has ushered in a new era of decentralized applications, with smart contracts being a key component. Smart contracts are self-executing contracts with the terms of the agreement directly written into code. While they promise to revolutionize various industries by enabling trustless transactions, they are also prone to security vulnerabilities that can have significant financial and reputational repercussions. This section provides a comprehensive overview of blockchain technology, smart contracts, their security issues, existing security analysis methods, and the integration of large language models (LLMs) with static analysis tools to enhance smart contract security.

3.1. Overview of Blockchain Technology and Smart Contracts

Blockchain technology, introduced by Satoshi Nakamoto in 2008 through the creation of Bitcoin, has revolutionized the way we perceive and implement decentralized systems [1]. A blockchain is a distributed ledger that maintains a continuously growing list of records called blocks, which are linked and secured using cryptographic techniques. Each block contains a cryptographic hash of the previous block, a timestamp, and transaction data, ensuring the immutability and integrity of the data stored on the blockchain [3].

Smart contracts, a pivotal innovation in blockchain technology, were popularized by the Ethereum platform introduced by Vitalik Buterin in 2014 [2]. Smart contracts are self-executing contracts with the terms of the agreement directly written into code. These contracts run on the blockchain, allowing for transparent, trustless, and automated execution of contractual agreements without the need for intermediaries [29].

The applications of smart contracts span various domains, including finance, supply chain management, healthcare, and governance [6]. By enabling programmable, self-enforcing agreements, smart contracts have the potential to significantly reduce costs, increase efficiency, and enhance security in transactions and processes. For example, in finance, smart contracts can automate complex transactions, reducing the need for intermediaries and lowering transaction costs [4]. In supply chain management, they can provide real-time tracking and verification of goods, improving transparency and reducing fraud [7]. In healthcare, smart contracts can enhance the security and privacy of patient data, streamline billing and claims processing, and ensure compliance with regulations [5].

The growing interest in blockchain and smart contracts is reflected in the increasing number of studies and implementations across various sectors. Researchers are continually exploring new ways to enhance the scalability, security, and functionality of blockchain systems [30], as well as addressing challenges such as energy consumption and regulatory compliance [31].

3.2. Security Issues in Smart Contracts

Despite their transformative potential, smart contracts are susceptible to various security vulnerabilities. Notable security incidents, such as the DAO attack in 2016, have
highlighted the risks associated with smart contract deployment [9]. The DAO attack exploited a re-entrancy vulnerability, allowing an attacker to drain 50 million USD worth of Ether from the decentralized autonomous organization (DAO) [11].

Common vulnerabilities in smart contracts include re-entrancy, integer overflow and underflow, unchecked return values, and access control issues [12]. These vulnerabilities can lead to significant financial losses and undermine trust in blockchain-based systems. For instance, re-entrancy attacks occur when a function makes an external call to another untrusted contract before resolving its state changes, allowing attackers to exploit this by re-entering the function and manipulating balances. Integer overflow and underflow vulnerabilities arise when arithmetic operations exceed the maximum or minimum size of the data type, potentially causing incorrect calculations and financial discrepancies [8].

Addressing these security issues is crucial for the widespread adoption and success of smart contracts. Security audits, formal verification, and the development of more robust coding practices are essential steps in mitigating these risks [13]. Continuous monitoring and updating of smart contracts are also necessary to adapt to new threats and vulnerabilities [32].

3.3. Existing Security Analysis Methods

Several approaches have been proposed to enhance the security of smart contracts. Static analysis tools, such as Oyente [11] and Mythril [33], analyze the bytecode of smart contracts to identify potential vulnerabilities before deployment. Formal verification methods, leveraging mathematical proofs, ensure that smart contracts behave as intended under all possible conditions [34]. Runtime monitoring and auditing by third-party services also provide an additional layer of security by detecting anomalies in contract execution [35].

3.4. Static Analysis Tool: Slither

Static analysis is a method of examining and analyzing computer software without executing the program. This technique is particularly useful in identifying security vulnerabilities in smart contracts, as it allows for comprehensive code coverage and early detection of flaws [14]. Static analysis tools analyze the source code or bytecode of smart contracts to detect potential security issues before deployment.

Slither [20] is one of the most prominent static analysis tools for smart contracts. Developed by Trail of Bits, Slither uses advanced techniques such as data-flow analysis, control-flow analysis, and taint analysis to identify vulnerabilities in Solidity smart contracts. It is widely adopted in both industry and academia for its robustness and effectiveness. For example, data-flow analysis in Slither tracks the flow of data through the program to detect if any sensitive data might be exposed or misused, while control-flow analysis aids in the understanding of paths that might be taken during execution to identify logical errors [21].

However, static analysis tools like Slither have limitations. They often struggle with accurately analyzing state variable dependencies and complex code structures, leading to incomplete vulnerability detection [21]. These limitations are particularly evident in the context of smart contracts that interact with external systems or have intricate state transitions. Improving the accuracy and completeness of static analysis tools is essential for enhancing the security of smart contracts [36].

3.5. Large Language Models

Large language models (LLMs), such as OpenAI’s GPT-4, have demonstrated remarkable capabilities in natural language understanding, code comprehension, and generation tasks [22]. These models are trained on vast amounts of text data and can generate human-like text, making them suitable for a variety of applications, including code analysis [23].

LLMs can understand and contextualize source code, identify patterns, and suggest improvements, making them valuable tools for enhancing static analysis techniques [24]. They exhibit advanced code comprehension capabilities that can be leveraged to address
the limitations of traditional static analysis tools. For instance, learning-distributed representations of code can provide insights into potential vulnerabilities by understanding the context and dependencies within the code that static analysis tools might miss [37].

Integrating LLMs with static analysis tools offers a promising approach to improving the accuracy and completeness of vulnerability detection in smart contracts. By leveraging the strengths of both technologies, we can enhance the security and reliability of blockchain-based systems [38]. This integration can lead to more intelligent and adaptive security tools that not only detect known vulnerabilities but also anticipate and mitigate new threats [39].

By providing a deeper understanding of the context and logic of smart contracts, LLMs can help bridge the gap between static analysis and dynamic analysis, offering a more comprehensive security assessment. This hybrid approach can significantly reduce the risk of deploying vulnerable smart contracts and contribute to the overall trust and robustness of blockchain ecosystems [40].

3.6. Few-Shot Learning

Few-shot learning (FSL) is a machine learning approach where a model is trained to recognize and generalize from a very limited number of examples [41]. This technique is particularly useful in scenarios where labeled data are scarce or expensive to obtain. In the context of smart contract security, FSL can be employed to enhance the performance of static analysis tools by enabling the model to learn from a small number of annotated vulnerabilities and generalize to unseen code structures [42].

Recent advances in FSL have demonstrated its effectiveness in various applications, including natural language processing and computer vision [43]. By incorporating FSL techniques into the analysis of smart contracts, we can improve the detection of rare or novel vulnerabilities that traditional static analysis tools might miss. This approach can be particularly beneficial in identifying zero-day exploits and emerging security threats in blockchain ecosystems [44].

3.7. Chain-of-Thought

Chain-of-Thought (CoT) is a reasoning approach that leverages the sequential and logical flow of thoughts to solve complex problems [45]. In the context of smart contract security, CoT can be used to enhance the capability of large language models (LLMs) in understanding and analyzing code by breaking down the reasoning process into smaller, manageable steps. This method allows for a more thorough examination of the code’s logic and dependencies, leading to more accurate identification of vulnerabilities [22].

CoT can be integrated with static analysis tools and LLMs to provide a comprehensive security assessment of smart contracts. By simulating the thought process of a human expert, CoT can help identify subtle and complex vulnerabilities that might be overlooked by automated tools [46]. This hybrid approach can significantly enhance the robustness and reliability of smart contract security analysis, contributing to the overall security of blockchain-based systems.

3.8. Previous Work on Improving Smart Contract Security

The literature on smart contract security reveals various vulnerabilities that have been exploited in high-profile attacks, leading to significant financial losses. For instance, the infamous DAO attack highlighted the vulnerability of re-entrancy, where an attacker could repeatedly call a function before previous executions were completed, draining funds from the contract [9,11]. Further studies have identified other common vulnerabilities, such as integer overflows and underflows, unhandled exceptions, and timestamp dependence [12,34]. Atzei et al. [12] provided a comprehensive survey of known vulnerabilities in Ethereum smart contracts, categorizing them based on their causes and effects. Bhargavan et al. [34] emphasized the criticality of formal verification techniques to ensure contract correctness and prevent security breaches.
To address these vulnerabilities, researchers have proposed several solutions. Bhargavan et al. [34] introduced a framework for formal verification of smart contracts, aiming to ensure the correctness of contract execution. Delmolino et al. [32] emphasized the importance of educating developers on secure coding practices for smart contracts. Additionally, tools like Oyente [11] and Mythril [33] have been developed to perform automated security analysis, identifying potential vulnerabilities in smart contract code. These tools utilize symbolic execution and other static analysis techniques to detect common security issues, providing developers with actionable insights to improve contract security.

With the background established, this paper will proceed to discuss the proposed methodology for integrating large language models with Slither to enhance its static analysis capabilities for smart contract security. Following the methodology, this paper will present experimental results, address potential limitations, and suggest future research directions.

4. Motivation
4.1. Function Dependency Analysis

Function dependency analysis is an analysis that explores the dependency relationships among functions. A smart contract is a stateful program that consists of state variables and functions. Some functions can use state variables in branching conditions to control the business logic. Some other functions can modify the state variable. Thus, one type of function dependency is based on the state variables read in branching conditions and written in functions. If function A reads a state variable in a branching condition and function B can write this state variable, then function A depends on function B.

Function dependency analysis is an important static analysis for both the testing and security analysis of smart contracts. When a contract is deployed, state variables cannot be updated arbitrarily. In other words, state variables are modified by executing functions that can modify them. Therefore, to test a function, certain functions need to be executed first to change the values of the state variables used in the branching conditions of the function. As a result, when executing this function after them, the branching conditions of this function are more likely to be satisfied; thus, more code can be covered. When the code coverage increases, the chance of detecting vulnerabilities is improved as well. Therefore, function dependency analysis can be applied to the fuzzing and symbolic execution of smart contracts for testing and security analysis.

To perform the function dependency analysis, the first step is to identify the state variables read in branching conditions and the state variables written by each function. Slither [20], a widely used static analysis tool in academia and industry, has the APIs to obtain the state variables read in conditions and the state variables written for a given function.

4.2. Limitations of Slither

However, studies show that Slither can fail to detect the state variables read in conditions (i.e., it has false negatives). Slither can also fail to distinguish branching conditions from other conditions. Below, we present four common cases where Slither fails to identify the state variables read in branching conditions.

4.2.1. False Negative 1

When a state variable is assigned to a local variable, and this local variable is used in a branching condition, the state variable is considered read in the branching condition. Slither fails to detect this.

In Contract FN1 shown in Listing 1, there is one function updateValue(). The state variable $a$ is assigned to a local variable $temp$ in Line 9. Then, $temp$ is used in the if condition in Line 10. Therefore, $a$ is considered read in the if condition in Line 10. However, Slither can not detect that $a$ is read in this if condition in Line 10.

```solidity
pragma solidity ^0.8.0;
contract FN1 {
    uint 256 a=0;
    uint256 value=0;
    constructor (){
        
    function updateValue(uint256 newValue) public {
        // false negative: a is not considered read in a branching condition by Slither
        uint256 temp=a;
        if (temp==10){
            value=newValue;
        }
    }
}
```

4.2.2. False Negative 2

When a function that reads a state variable is called in a branching condition, this state variable should be considered read in this condition of the caller. Nevertheless, Slither fails to recognize that this state variable is read in the branching conditions of the caller function.

For example, in Contract FN2 shown in Listing 2, there are two functions, `getA()` and `updateValue()`. `getA()` reads state variable `a` and is used in the `if` condition of the function `updateValue()` in Line 12. State variable `a` is used in this `if` condition to compare with a constant 10. Slither can not identify that `a` is read in a branching condition in function `updateValue()`.


```solidity
pragma solidity ^0.8.0;
contract FN2 {
    uint256 a=0;
    uint256 value=0;
    constructor (){
    }
    function getA () public returns (uint256){
        return a;
    }
    function updateValue(uint256 newValue) public {
        // false negative: a is not considered read in a branching condition by Slither
        if (getA()==10){
            value=newValue;
        }
    }
}
```

4.2.3. False Positive 1

When a function returns a state variable of the type `bool`, this state variable is read in a condition in this function according to Slither. However, this state variable is actually not read in a branching condition.

In Contract FP1 in Listing 3, Function `getFlag()` returns `flag`, which is of the type `bool`. Slither considers the state variable `flag` read in a condition of Function `getFlag()`. However, there is no branching condition in Function `getFlag()`. Therefore, Slither treats a variable of type `bool` in the `return` statement as a condition, which can result in a false positive.


```solidity
pragma solidity ^0.8.0;
contract FP1 {
    bool flag=false;
    constructor (){
    }
    function getFlag () public returns(bool){
        //false positive: flag is considered as read in a condition by Slither
        return flag;
    }
}
```
4.2.4. False Positive 2

When a function returns a conditional expression, the state variables used in the conditional expression are considered as the state variables read in conditions by Slither. In other words, Slither treats conditional expressions in the return statement as conditions. However, these conditions are not the branching conditions in a function used to control the business logic of the function.

For instance, in contract FP2 in Listing 4, a and value are two state variables. They appear in a conditional expression $a == value$ in the return statement. Slither reports that $a$ and $value$ are the state variables read in conditions of the function `getBoolValue()`.


```solidity
pragma solidity ^0.8.0;
contract FP2 {
    uint256 a = 0;
    uint256 value = 0;
    constructor (){}
    function getBoolValue () public returns ( bool ){
        // false positive: a and value are considered as read in a condition by Slither
        return a == value;
    }
}
```

4.3. Proposal to Perform a Function Dependency Analysis

We propose a method to perform a function dependency analysis based on GPT-4o and Slither. GPT-4o has shown remarkable capabilities in understanding and programming code. The advantage of GPT-4o is that it can analyze code like an expert and thus can be used to perform code analysis beyond the limits of traditional engineering tools. However, it sometimes behaves like a layman who has limited knowledge of the code being analyzed, which is known as a hallucination issue. Slither is a widely used static analysis tool that produces results deterministically and never generates anything that does not make any sense. This is the advantage of Slither. As shown in the subsection immediately above, however, Slither can fail in some cases. By combining GPT-4o and Slither, we can take advantage of both GPT-4o and Slither.

We formulate the function dependency analysis as the data-refining process. We first prepare for the initial dependency data. Then, we refine the initial dependency data. The key insights are as follows:

- Slither can not only provide dependency data but also other information that can be used to provide additional information for further analysis.
- GPT-4o fits the job of refining data, as it is good at analyzing code like a human. The hallucination issue can be alleviated by carefully designing the refining process and providing additional context information to limit the evaluation scope.

We first apply Slither to collect function dependency data and other information like state variables and modifiers. Next, we utilize GPT-4o to refine the function dependency data through three different roles: the checker, evaluator, and verifier. Given a piece of data of a function, some other data, and function code, the checker initially checks whether the function is required to be closely examined or not. If not, then the refining process stops. Otherwise, the evaluator further evaluates the function data through a conversation with the verifier. Receiving the function data to be evaluated, the evaluator first examines them. Next, the evaluator updates the function data if it finds them to be incorrect, or it accepts them if it agrees with the given data. Then, the function data are sent to the verifier to verify. If the verifier determines that the data should be allowed to pass, the function data are then accepted, and the refining process terminates. Otherwise, the verifier sends back the feedback to the evaluator, which then evaluates the function data again while being aware of the feedback. The updated function data then again go to the verifier. The conversation ends when the function data pass the verifier or the conversation length...
reaches a user-defined limit. Note that the conversation length refers to the number of evaluation–verification cycles.

The novelty of our proposal is that we define the dependency analysis as the data-refining process, and we create three different roles of GPT-4o to form a refining process to refine the data. Our proposal is named Sligpt, which is the concatenation of “sli” and “gpt”. “Sli” is the first three letters from “Slither”, and gpt is the first three letters from “GPT-4o”.

5. Approach

We present our approach, Sligpt, to perform a function dependency analysis. Sligpt employs GPT-4o to refine the function dependency data produced by Slither. The function dependency data are the state variables read in branching conditions and the state variables written in functions. Sligpt is designed to refine the state variables read in branching conditions while keeping the state variables written without evaluation. This design decision is based on the observation that Slither can detect the state variables written correctly and the general rule of selecting a tool for an engineering task. If the data can be correctly obtained using a traditional engineering tool, then it is better to use the traditional tool instead of a learning model that is not deterministic.

5.1. Architecture

Figure 1 presents the architecture of Sligpt. It has three main components: the Slither data provider, function RW collector, and GPT-4o analyzer. The components o the Solidity file and function RW denote the input and output entities, respectively. The labels of the arrows connecting the components show the data flowing between the components. The Slither data provider provides initial function dependency data for further refinement and other data that may serve as information to help the GPT-4o analyzer to reduce instances of hallucinations. the function RW collector deals with the state variables read in the branching conditions (R) and the state variables written (W) for each function. It relies on the GPT-4o analyzer to examine R and outputs the RW data for each function. The GPT-4o analyzer investigates R to refine it and returns to the function RW collector. The GPT-4o analyzer refines R through three roles: the checker, evaluator, and verifier. The checker acts as a filter to filter out the functions that do not need to be refined. The evaluator and verifier form the evaluation–verification loop to refine R.

5.2. Workflow

In this subsection, we show the end-to-end workflow of Sligpt, starting with the given Solidity file of a contract and ending with the collected function dependency data of the contract. Initially, the Slither data provider and the GPT-4o analyzer receive the source code of the given Solidity file, as indicated by the arrow numbered 1. Next, the Slither data provider invokes Slither to collect for each function the invoked modifiers, the state variables read in the branching conditions (i.e., R), the state variables written (i.e., W), and all the state variables defined in the contract. These collected data are the static analysis data that go to the function RW collector, as indicated by the arrow numbered 2. Then, the function RW collector drives the GPT-4o analyzer to refine R for each function. Note that Sligpt is designed to refine R instead of W, as Slither can correctly identify W.

For each function, the function RW collector obtains the RW related to the function and other data (presented as ... in Figure 1) from the static analysis data, including all state variables and the modifiers invoked in the function. Then, the collector calls the GPT-4o analyzer to refine R by providing the R and other data (see the arrow numbered as 3). When the GPT-4o analyzer returns the refined R, the function RW collector uses the returned R and the original W as the final RW of the function. When all the functions are considered, it outputs a collection of function RW, as indicated by the arrow numbered 9.
The GPT-4o analyzer, to save on costs, avoids refining the functions that do not need to be examined (the reasons are given in Section 5.3.1). Therefore, the checker is employed to filter out such functions. When a function is filtered out, the GPT-4o analyzer returns the received $R$ directly back to the function RW collector, shown by the arrow numbered 4. Otherwise, the $R$ of the function and other data are sent to the evaluator (see the arrow numbered 5) to refine.

The evaluator evaluates the $R$ to determine whether to accept it or not. If the evaluator does not accept it, the evaluator then provides the updated $R$ that it thinks is correct. Next, the evaluator passes the accepted or updated $R$ along with other data to the verifier for verification, as shown by the arrow numbered as 6. If the verifier agrees with the received $R$, the GPT-4o analyzer returns this $R$ (see the arrow numbered as 8). Otherwise, the verifier provides feedback to the evaluator (see the arrow numbered as 7). When the evaluator receives the feedback, it re-evaluates $R$ by considering the feedback. At this point, a loop is formed between the evaluator and the verifier. This evaluation–verification loop is repeated until the verifier agrees with the received $R$.

5.3. GPT-4o Analyzer

The GPT-4o analyzer is the core component of Sligpt to refine the given function dependency data. It is designed with consideration of the costs, the hallucination issue, and the indeterministic nature of LLMs. We design the analyzer based on chain-of-thought prompting, which is a type of few-shot learning with reasoning steps (thoughts) available.

5.3.1. Checker

The role of the checker is to initially examine the function to determine the necessity of further refining the process. This role is introduced while considering the cost. The GPT-4o analyzer is not free of charge. The unnecessary invoking of the GPT-4o analyzer should be recognized and reduced. Therefore, the checker is designed to reduce the unnecessary invoking of the GPT-4o analyzer.

The key insight to creating the checker is that the Slither data provider can provide correct dependency data except for the failure cases mentioned in the Motivation section. Therefore, our idea for designing the checker is to examine the function code to determine whether it has the code patterns resulting in the failure cases. Recall that there are four code patterns causing the failure cases:

![Figure 1. Architecture of Sligpt. $R$: the state variables read in the branching conditions of a function. $W$: the state variables written in a function. ...: denotes the data providing additional information based on static analysis data (e.g., state variables, modifiers). Static analysis data: include all the defined state variables and the data of each public or external function: modifiers, the state variables read in branching conditions ($R$), and the state variables written ($W$). Note that Sligpt currently does not refine $W$, as Slither can correctly identify $W$.](image-url)
1. Having a local variable in a branching condition that has the value expressed in terms of a state variable.
2. Having a function call in a branching condition such that the invoked function reads a state variable.
3. Having a return statement that returns a state variable of type bool.
4. Having a return statement that returns a boolean expression involving state variables.

We design the checker as the prompt design. The design is based on the four code patterns. Figure 2 visualizes the prompt design of the checker. The checker prompt first defines the constraints that are used to capture the code patterns mentioned above. As long as one of the constraints is satisfied, the function should be further checked.

Code: contract code
Question: is function function name required to be checked?
Examples:

Code: contract code: contract FN1 { uint256 a=0; uint256 value=0; function updateValue(uint256 newValue) public { uint256 temp=a; if (temp==10){ value=newValue; } }}

Question: is function updateValue(uint256) required to be checked?
Answer: Function updateValue(uint256) has defined a local variable "temp" that has the value expressed by the state variable "a". Then, this local variable is used in the branching condition "temp==10" from an if statement. So, constraint 1) is satisfied; thus, this function needs to be checked. The answer is yes.

Figure 2. Prompt design of the checker. ...: denotes the data providing additional information (e.g., state variables, modifiers). The prompt has input parameters: contract code, function name, and others. The prompt content includes the constraints to identify particular code patterns, code a question related to the code, and a few examples showing how to answer questions.

The next part of the checker prompt content depends on the provided inputs at runtime: the Solidity source code of a contract, a function defined in this contract, and others. Others include the defined state variables and modifiers. They are optional and can be added to the prompt after the constraint description as an extra scope limit (not presented in the prompt design). The contract code is provided instead of the function code. The reason is that the function code may involve other functions and modifiers. Identifying function code needs to correctly collect all the function calls and modifiers and then their corresponding code. This process requires careful engineering. In addition, the token limit of GPT-4 is 128,000 tokens. Directly giving the contract code does not cause the input tokens to reach the limit.
At the end of this prompt, a few examples are provided to show the steps to conclude whether a constraint is satisfied or not. Since the task is about reasoning whether a piece of code satisfies the unseen constraints, we thus provide examples to demonstrate how to perform such a task. For each constraint, we provide at least one example. Take the example shown in Figure 2, the question is, "does the function updateValue(uint256) need to be checked?" The steps necessary before reaching a conclusion are as follows:

(a) A local variable temp is defined in function updateValue(uint256).
(b) State variable a is assigned to temp.
(c) temp is used in a condition from an if statement.
(d) Constraint 1 is thus satisfied.
(e) The answer is yes.

By defining the constraints to detect the code patterns and giving examples to demonstrate how to evaluate if the given piece of code satisfies one of the constraints, the prompt can be prompted to GPT-4o to answer questions related to the given function defined in the given contract code.

5.3.2. Evaluator

The role of the evaluator is to evaluate the function code to determine whether the given data of the function are acceptable or not. The evaluation process is designed as the iteration of the evaluation–verification cycles due to the issue of hallucination of LLMs. In this section, we first show the design of the prompt of determining the acceptance of the given data. Then, we present the prompt during the iteration process.

To evaluate the given data, we identify a list of rules to detect the state variables read in branching conditions (R). A few examples are provided to demonstrate how to identify the state variables based on the given rules. The rules and examples are the main elements of the initial evaluator prompt. The design of the initial evaluator prompt is shown in Figure 3. Note that the prompt of the evaluator grows along with the evaluation–verification loops. Therefore, we have the initial prompt.

As shown in Figure 3, there are five rules defined to detect R. These five rules encompass the general or common patterns of how R appears in the code. We admit that we may miss some patterns that we have not observed so far. However, the evaluator can be easily adapted by adding more rules.

To further assist the evaluation task, we provide at least one example for each rule so that the GPT-4o can learn and know how to evaluate. Each example has the contract code, the question based on a function of the code, and the answer to the question. Take the example shown in Figure 3. The contract FN1 has a function named updateValue with a parameter of type uint256. The question is whether the given data about the function are correct. The answer formats are also provided for the question. As for the answer, it shows the steps of how to reach the final answer.

The initial evaluator prompt also includes the contract code that contains the function to be analyzed and the question about the function. The question is not directly to ask the evaluator to identify R for the given function. Instead, the question is to evaluate the function code based on the rules first and then check if the given data are acceptable based on the evaluation.

To reduce the hallucination issue, the evaluator is designed to enable multiple iterations, such that in each iteration the feedback of the evaluation is provided. The feedback (from the verifier, which is presented in Section 5.3.3) can provide some insight, helping the evaluator to reason. In terms of the evaluator, in each iteration, it has a question about R, obtains the accepted R from the answer to the question, and receives the feedback about R.

As the past iterations are important for the evaluation in the current iteration, at the beginning of each iteration, the prompt is designed to be accumulated such that it can have a comprehensive record of this history. The prompt elements of the evaluator are visualized in Figure 4. When t (iteration loop) = 0, the prompt is the initial prompt, which is presented in Figure 3. It consists of rules (Rules), rule demonstration examples (Examples), contract
code (Code), and the question (Q₀(R, F)) about a function (F) and R of the function. In this iteration, the answer (A₀) to the question is obtained, and the feedback (FB₀) is received from the verifier. When t = 1, the question Q₁(R, F) is based on the R that is accepted when t = 0. The prompt in this iteration is thus the concatenation of Rules, Examples, Code, Q₀(R, F), A₀, FB₀, and Q₁(R, F). This process is continued until the maximum iteration limit is reached.

**Rules:**
Here are the rules to identify the state variables read in branching conditions of function A. Please keep in mind that we only target the conditions in assert, require, if, for, while, and ternary operation statements. In other words, you must ignore the conditions in the return statement in function A. A state variable is read in a branching condition of function A if one of the four rules below is met:

1) It is read or used in a condition of function A.
2) It is used to express the value of a local variable, which is then read or used in a condition of function A.
3) It is read or used in a function and this function is invoked in a condition of function A.
4) It is read or used in a condition and this function is called in function A.
5) It is read or used in a condition in the function modifiers invoked in function A.

**Examples:**

**Code:**
```solidity
contract FN1 {
    uint256 a=0;
    uint256 value=0;
    function updateValue(uint256 newValue) public {
        uint256 temp=a;
        if (temp==10){
            value=newValue;
        }
    }
}
```

**Answer:** Function updateValue(uint256) has only one condition "temp==10". It is in the if statement. "temp" is a local variable that has the value expressed by the state variable "a". Therefore, "a" is the state variable read in branching conditions of function updateValue(uint256), which meets rule 2). So, the given data are correct and acceptable. Answer: "accept".

**Figure 3.** The initial prompt of the evaluator. ... denotes other data (e.g., state variables, modifiers). The initial prompt has the following input parameters: contract code, function name, the reads of state variables in the branching conditions (R), and others. The prompt content includes the rules to find R, examples of how to find R, the source code of the contract defining the input function, and the question related to the function and R.
5.3.3. Verifier

The verifier is included as a mechanism to reduce the impact of hallucinations. It further examines the given R accepted by the evaluator. If the reasoning process of the evaluator has inconsistency, the verifier then investigates the R from a different perspective, which is supposed to reveal the inconsistency. If the verifier finds no issues, then it allows the given R to pass so that the GPT-4o analyzer stops the refining process and returns the R. Otherwise, the verifier delivers the investigation results to the evaluator as feedback, such that the evaluator will re-evaluate R.

The verifier is designed to verify the given R accepted by the evaluator. In this case, the verifier acts as an expert in function data dependency analysis. Generally, an expert is expected to know the parts that are more likely to go wrong. Therefore, our idea to verify the given R is to analyze the points where R is likely wrongly identified. Hence, we design the verification as the process used to answer a list of questions regarding those points.

We raised six questions based on the given R of the target function based on the points that the Slither data provider and the evaluator are likely to make mistakes. The verifier verifies the given R by answering these questions one by one. When all the answers to these questions are “no”, the verifier accepts the R and allows it to pass. Otherwise, the verifier provides the reasons for the questions, the answers to which are not “no”. These reasons are the feedback sent to the evaluator.

Figure 5 shows an example of how to verify the given data for the function `updateValue(uint256)` in contract FN1. Questions (a) and (b) examine the case that the state variables are used in a function, while this function is then invoked in `updateValue(uint256)`. Question (c) checks the state variables used in the modifiers. Question (d) is to check if a state variable is considered because it is used in the return statement. Question (e) is to evaluate if all the state variables in the R are correctly identified. If each state variable in the R has the related conditions, then all the state variables in the R are correctly identified. Note that the related conditions of a state variable svar are the branching conditions that svar is identified as read in them. For example, if v is a state variable and there is a function F that has a local variable temp having the value expressed by v and a branching condition `temp > 10`, then v is identified as read in a branching condition of the function F,
and \( \text{temp} > 10 \) is a related condition of \( v \). Question (f) examines whether a state variable is missed when it is used to express the value of a local variable that is then used in a branching condition.

**Code:**
```solidity
Contract code: contract FN1 {
    uint256 a=0;
    uint256 value=0;
    function updateValue(uint256 newValue) public {
        uint256 temp=a;
        if (temp==10){
            value=newValue;
        }
    }
}
```

**Question:**
Here are the suspicious data about the state variables read in branching conditions in function `updateValue(uint256)`:

- `{"updateValue(uint256)":{"state_variables_read_in_BC":["a", "value"]}}`

a) Do the given data miss the state variables that are read or used branching conditions in function `updateValue(uint256)`?
b) Do the given data miss the state variables that are read or used in branching conditions in a function and this function then is invoked in function `updateValue(uint256)`?
c) Do the given data miss the state variables that are read or used in a branching condition in the modifiers invoked in function `updateValue(uint256)`?
d) Do the given data contain the state variables read in the conditions in the return statements?
e) Is there a state variable in `["a", "value"]` having no related conditions function `updateValue(uint256)` or in the invoked functions or modifiers?
f) Do the given data miss the state variable that is used to express the value of a local variable and then this local variable is used in a branching condition in function `updateValue(uint256)`?

Please answer each question one by one. When the answers to all the questions are no, then reply “pass”; otherwise, please reply with the reasons for the questions that the answers are yes.

**Answer:**
a) Function `updateValue(uint256)` has only one condition "temp==0" in the if statement. As the value of "temp" is not expressed by a function call but a state variable, this condition hence does not contain a function call. Hence, there is no state variable that is read in a function that is invoked in a branching condition in the function `updateValue(uint256)`. So, the answer is no.
b) Function `updateValue(uint256)` does not invoke any function call inside itself. Therefore, there is no state variable read in conditions of another function that is called in function `updateValue(uint256)`. So, the answer is no.
c) Function `updateValue(uint256)` has no modifier invoked in it. Therefore, there are no state variables read or used in a branching condition in the modifiers invoked in function `updateValue(uint256)`. So, the answer is no.
d) This function has no return statement. Hence there is no state variables used in a condition in the return statement of function `updateValue(uint256)`. Therefore, the answer is no.
e) The related condition of a state variable is the condition through which the state variable is identified based on the given rules. Function `updateValue(uint256)` has only one condition "temp==10" in the if statement, in which "temp" is a local variable that is expressed by the term using the state variable "a". So, temp==0 is related to "a", "value" is not read in any condition in function `updateValue(uint256)` or the invoked functions or modifiers according to the rules, and hence it is the state variable that has no related conditions. Therefore, the answer is yes.
f) Since not all the answers to the questions are no, the reply should be "not pass". Answer: "not pass".

**Figure 5.** Example showing how to verify a given \( R \) for a function. The verification process is about answering a series of questions. When all the answers are “no”, the verification passes.

As shown in Figure 5, for each question, the steps to reach the answer are provided. The answers to the first five questions are “no”. However, the answer to the last question is “yes”. Therefore, the given data fail to pass the verification. The final answer is “not pass”.

The prompt design of the verifier is shown in Figure 6. It includes the rules used to identify \( R \), examples of how to verify, and the questions related to \( R \), the target function, and
a series of verification questions. Note that the verifier does not depend on the evaluator. It only sees the given data and verifies them. Therefore, the identification rules should also be provided to help answer the questions.

In addition, the number of questions can be reduced based on other data. For example, if the function has no modifier, then question (c) can be removed. If all the state variables defined in a contract are in the given data, then questions (a), (b), (c), and (f) can be ignored, as it is impossible to miss any state variables.

We provide the rules to identify state variables read in branching conditions:

A state variable is read in a branching condition of function A if one of the four rules below is met:

1. It is read or used in a condition of function A.
2. It is used to express the value of a local variable, which is then read or used in a condition of function A.
3. It is read or used in a function and this function is invoked in a condition of function A.
4. It is read or used in a condition and this function is called in function A.
5. It is read or used in a condition in the function modifiers invoked in function A.

Next, we present a few examples to show how to verify:

Examples:

Code: contract code
Question:

Now, here are the suspicious data about the state variables read in branching conditions of the function function name: R.

a) Do the given data miss the state variables that are read or used branching conditions in function function name?

b) Do the given data miss the state variables that are read or used in branching conditions in a function and this function then is invoked in function function name?

d) Do the given data contain the state variables read in the conditions in the return statements?

c) Do the given data miss the state variables that are read or used in a branching condition in the modifiers invoked in function function name?

e) Is there a state variable in R having no related conditions function function name or in the invoked functions or modifiers?

f) Do the given data miss the state variable that is used to express the value of a local variable and then this local variable is used in a branching condition in function function name?

Please answer each question one by one. When the answers to all the questions are no, then reply "pass"; otherwise, please reply with the reasons for the questions that the answers are yes.

Figure 6. Prompt design of the verifier. ...: denotes other data (e.g., state variables, modifiers). A concrete demonstration example is given in Figure 5.

5.4. Discussion of How to Handle Hallucinations

Large language models (LLMs) suffer from hallucinations [47–49], a phenomenon where AI generates convincing but nonsensical answers. OpenAI acknowledges that the responses generated by ChatGPT may sound plausible but be nonsensical or incorrect.

Our approach is designed with an awareness of the hallucination issue. While we cannot entirely overcome the hallucination problem, we have implemented several strategies to minimize its occurrence:

1. Refinement of dependency data: We refine the dependency data obtained through a static analysis tool rather than rely on an LLM to directly generate this data.
2. Multiple evaluation–verification loops: We employ multiple evaluation–verification loops in the refining process instead of one loop.
3. Minimizing LLM dependency: We reduce the parts of the process that require an LLM for refinement or evaluation. For example, we only refine the reads of the state
variables and evaluate the reads for functions that are likely to present failure patterns identified by Slither.

4. Providing context information: We supply additional context information, referred to as “other data” in this paper. This includes providing a list of state variables that the LLM can consider and the modifiers of a function. All this information can be obtained along with the dependency data.

By implementing these strategies, we aim to reduce the impact of hallucinations and enhance the reliability of our method.

6. Evaluation

In this section, we evaluate Sligpt based on its performance when identifying the dependency data.

Dataset. To better evaluate Sligpt, we collected 10 Solidity smart contracts, each containing functions where the state variables read in branching conditions are hard to identify. These 10 contracts include all the cases mentioned in Section 4. (Please note that we can consider many more contracts; however, they either do not present the failure patterns mentioned in the Motivation section, or they repeat the same patterns in the selected 10 contracts). There are a total of 73 public or external functions (i.e., user-callable functions) without considering the constructor and the public functions of the state variables (note that public state variables are treated as public functions). On average, each contract has seven functions, indicating that the contracts selected are not simple.

Tools for comparison. We compare Sligpt with Slither (version 0.9.6), a state-of-the-art static analysis tool, using Solidity smart contracts. Slither converts Solidity source code to an intermediate presentation and then performs different analyses on the intermediate presentation. It provides APIs to collect the state variables read and written in functions. We also compare Sligpt with GPT-4o(version gpt-4o-2024-05-13), one of the most popular and powerful LLMs. It is trained on code and can understand code syntax and semantics like a human. We prompted GPT-4o with the proper requests. We admit that the content of the prompts may impact the performance of GPT-4o. However, we attempted to prepare the prompts based on content that was similar to the prompts used in Sligpt to reduce the impact of the prompts.

Metrics. We used the common metrics [50] to measure the performance based on the precision, recall, accuracy, and F1 score. In this paper, precision reflects how many state variables read in branching conditions are correctly identified out of the reported state variables, recall describes the actual state variables read in the branching conditions (i.e., the ground truth), and accuracy reflects the combination of the reported state variables and the ground truth. The F1 score reflects a balance between precision and recall.

Experiments. We run Slither to obtain the state variables read in the branching conditions and the state variables written for each function of 10 contracts. As Slither is a deterministic tool that always produces the same results with the same input, we only run it once. Due to the non-deterministic and unpredictable nature of LLMs, we prompt GPT-4o and run Sligpt five times. To avoid overloading GPT-4o, we send requests every 20 s.

Result collection. We first manually identify the state variables read in branching conditions and written in each function in the dataset as the ground truth. Then for each function, we collect the state variables read in the branching conditions and the state variables that can be written in it for the three tools. Finally, we compute the values of the metrics we use to measure the performance on the state variables read in the branching conditions, as identifying them is challenging. For a function, given a list of reported state variables read in the branching conditions and a set of the actual state variables read in the branching conditions of this function, we count the number of successfully identified state variables, the number of state variables that fail to be detected, and the number of state variables that are reported but are not expected (i.e., not the state variables read in the branching conditions). These three numbers denote the numbers of true positives (TPs),
false-negatives (FNs), and false positives (FPs), respectively. For each function, we collect these three numbers. Then, we sum the corresponding numbers across all the functions. Finally, we compute the F1 score. Table 1 presents the values.

Table 1. Performance metrics comparison (note that Slither runs once, as it is a deterministic tool).

<table>
<thead>
<tr>
<th>Tool</th>
<th>Times</th>
<th>Precision</th>
<th>Recall</th>
<th>Accuracy</th>
<th>F1 Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>Slither</td>
<td>1st</td>
<td>75/85</td>
<td>75/89</td>
<td>75/99</td>
<td>0.86</td>
</tr>
<tr>
<td></td>
<td>2nd</td>
<td>80/109</td>
<td>80/89</td>
<td>80/118</td>
<td>0.81</td>
</tr>
<tr>
<td></td>
<td>3rd</td>
<td>80/112</td>
<td>80/89</td>
<td>80/122</td>
<td>0.79</td>
</tr>
<tr>
<td></td>
<td>4th</td>
<td>80/112</td>
<td>80/89</td>
<td>80/121</td>
<td>0.80</td>
</tr>
<tr>
<td></td>
<td>5th</td>
<td>80/110</td>
<td>80/89</td>
<td>80/119</td>
<td>0.80</td>
</tr>
<tr>
<td></td>
<td>1st</td>
<td>85/93</td>
<td>85/89</td>
<td>85/97</td>
<td>0.93</td>
</tr>
<tr>
<td></td>
<td>2nd</td>
<td>86/99</td>
<td>86/89</td>
<td>86/102</td>
<td>0.91</td>
</tr>
<tr>
<td></td>
<td>3rd</td>
<td>85/94</td>
<td>85/89</td>
<td>85/98</td>
<td>0.93</td>
</tr>
<tr>
<td></td>
<td>4th</td>
<td>88/95</td>
<td>88/89</td>
<td>88/96</td>
<td>0.96</td>
</tr>
<tr>
<td></td>
<td>5th</td>
<td>85/93</td>
<td>85/89</td>
<td>85/97</td>
<td>0.93</td>
</tr>
</tbody>
</table>

In Table 1, the numerators represent the counts of the correctly identified reads of state variables. For the Precision column, the denominators are the total counts of the reads of state variables in the branching conditions reported by the tools. For the Recall column, the denominators are the total counts of the reads of the state variables actually occurring in the branching conditions (i.e., the ground truth). In the Accuracy column, the denominators are the union of the reads reported by the tools and the ground truth.

Result analysis. Figure 7 visualizes the metric data for the three tools after averaging the five data points for GTP-4o and Sligpt. Figure 7 indicates that Sligpt has an accuracy of about 0.88; the highest. Slither has an accuracy of 0.76, while the accuracy of GTP-4o is about 0.67. When it comes to the F1 score, Sligpt achieves an accuracy of about 0.93; the highest as well. Slither and GPT-4o have accuracies of 0.86 and 0.8, respectively.

![Figure 7](image.png)

Figure 7. Column chart visualizing the metric data in Table 1 for Slither, GPT-4o, and Sligpt. Note that the metric data of GPT-4o and Sligpt are averaged across five data points.

Generally, GPT-4o has the lowest performance. Although GPT-4o can identify more state variables read in branching conditions than Slither, as shown in Table 1 (GPT-4o correctly identifies 80 state variables read in branching conditions, while Slither only identifies 75 state variables), GPT-4o can report many more state variables that are not read in the branching conditions compared to Slither (see the denominators in the Precision column in Table 1). Sligpt, by taking advantage of Slither and GPT-4o, manages to identify even more state variables correctly than GPT-4o, but it also reduces the number of false positives.
Discussion. We notice that Sligpt is quite unstable. This instability is demonstrated by
the data in the tables and is due to the unpredictable nature of GPT-4o.

Manually, we can make Sligpt correctly identify all state variables read in branching
conditions without false positives for each contract at least once. But if we run it multiple
times, Sligpt can have false positives and false negatives as well.

GPT-4o may also produce nonsensical responses that can impact Sligpt. Mostly, GPT-
4o can learn from the given examples about how to analyze and then apply the analysis
patterns like an expert. However, it can still fabricate data. For example, for a function with
the declaration `function join(address urn, uint wad) public note`, which has the modifier “note”,
one response from GPT-4o can be as follows:

```
Function join(address,uint) uses the modifier ‘auth’, which contains a require
statement with the branching condition wards[msg.sender] == 1. This condition
reads the state variable wards.
```

In this response, it shows that GPT-4o considers `auth` as the modifier of the function
`join(address,uint)`. However, only `note` is the modifier of the function `join(address,uint)`.

Another response is shown in the box below:

```
Function join(address,uint) has the modifier auth invoked in it because
‘auth’ appears in the function declaration ‘function join (address urn, uint wad)
public note auth’. The modifier reads a state variable ‘wards’.
```

This response goes even further. GPT-4o provides the wrong function declaration
`function join(address urn, uint wad) public note auth` as evidence to support its statement that
`auth` is the modifier of the function `join(address,uint)`.

7. Discussion

This study introduces a methodology that integrates GPT-4o with Slither to analyze
the function dependency of Solidity smart contracts. This section discusses the limitations
and future directions of the proposed methodology.

7.1. Limitations

Despite the promising results, the proposed methodology has several limitations.
First, the three roles of the GPT-4o analyzer are derived from the four failure cases
of Slither. The checker scrutinizes the code based on constraints associated with these
failure cases. The evaluator assesses the code according to the rules formulated from the
code patterns leading to the failure cases. The verifier confirms the findings by answering
questions centered around the failure cases. We identified the four failure cases of Slither
based on a manual evaluation of hundreds of smart contracts. However, it is possible that
Slither has other failure cases that we did not observe. Consequently, the GPT-4o analyzer’s
reliance on these four failure cases might be insufficient.

Second, Sligpt assumes that the state variable writes detected by Slither are accurate.
During our manual evaluation to identify the four failure cases, we observed that Slither
accurately detects state variable writes within functions. Therefore, Sligpt is designed
without refining the state variable writes. Nonetheless, our manual evaluation is not
exhaustive, and Slither might occasionally fail to correctly identify state variable writes.

7.2. Vision for the Future of Smart Contract Security

Data dependency analysis stands as a cornerstone in the realm of static analysis, crucial
for the testing and security examination of smart contracts. Tools such as ILF [15]
and SmartTest [16] leverage dependency data to train models for fuzzing and symbolic
execution, respectively. Similarly, Smartian [17] utilizes dependency data to bolster the
effectiveness of fuzzing in smart contract analysis. SmartExecutor \[18\] employs dependency data to guide symbolic execution, thereby enhancing code coverage. Additionally, Sailfish \[19\] relies on dependency data to identify state inconsistency bugs. The integration of large language models (LLMs) with static analysis tools presents a promising avenue for augmenting software security. By refining the accuracy and comprehensiveness of vulnerability detection, this method can mitigate the risk of security breaches and financial losses associated with smart contracts.

Looking ahead, the continuous advancement of LLMs and their amalgamation with sophisticated static analysis tools is anticipated to yield even more robust and comprehensive security analysis frameworks. These developments will be pivotal in ensuring the security and reliability of smart contracts and other blockchain applications, thereby fostering greater adoption and trust in blockchain technology. By enhancing the precision of data dependency analysis, these innovations will indirectly and significantly improve the outcomes of software security analysis, contributing to a safer and more secure digital ecosystem.

8. Conclusions

This research presents an innovative approach called Sligpt to perform data dependency analyses by integrating GPT-4o with Slither, a static analysis tool. Sligpt performs the dependency analysis through the process of refining the given dependency data. The given dependency data are collected using Slither, while the refining process is completed by the multiple roles of GPT-4o. The empirical evaluation reveals that Sligpt achieves significant improvements in dependency analyses.

Sligpt can be utilized for various downstream tasks, including taint analyses, fuzzing, guided symbolic execution, and the construction of machine-learning models. For instance, Sligpt can be directly applied to SmartExecutor for graph construction used to guide the symbolic execution process. This demonstrates Sligpt’s potential to significantly enhance the effectiveness of these tasks by improving the data dependency analysis.

Our research makes several significant contributions to the field of program analysis for smart contracts. Firstly, we developed a detailed methodology for integrating GPT-4o with the Slither static analysis framework to perform data dependency analysis. Secondly, through rigorous empirical evaluation using a curated dataset of Ethereum smart contracts, our approach demonstrated substantial improvements in precision, recall, and overall analysis depth when compared to Slither alone. These findings underscore the promise of combining LLMs with static analysis tools to fortify the testing and security analysis of smart contracts, paving the way for more secure and reliable blockchain applications.


Funding: This research received no external funding.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.


Conflicts of Interest: The authors declare no conflicts of interest.

References


31. Zohar, A. Bitcoin: under the hood. Commun. ACM 2015, 58, 104–113. [CrossRef]


Disclaimer/Publisher’s Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.