PROMFUZZ: Leveraging LLM-Driven and Bug-Oriented Composite Analysis for Detecting Functional Bugs in Smart Contracts

Xingshuang Lin*✦, Qinge Xie†✦, Binbin Zhao*(✉)‖, Yuan Tian‡, Saman Zonouz†, Na Ruan§, Jiliang Li¶, Raheem Beyah†, Shouling Ji*
*Zhejiang University, †Georgia Institute of Technology, ‡University of California, Los Angelos, §Shanghai Jiaotong University, ¶Xi'an Jiaotong University, ‖Engineering Research Center of Blockchain Application, Supervision And Management (Southeast University), Ministry of Education
E-mails: cs.xslin@zju.edu.cn, qxie47@gatech.edu, binbinz@zju.edu.cn, yuant@ucla.edu, szonouz6@gatech.edu, naruan@cs.sjtu.edu.cn, jiliang.li@xjtu.edu.cn, rbeyah@coe.gatech.edu, sji@zju.edu.cn

***Abstract*—Smart contracts are fundamental pillars of the blockchain, playing a crucial role in facilitating various business transactions. However, these smart contracts are vulnerable to exploitable bugs that can lead to substantial monetary losses. A recent study reveals that over 80% of these exploitable bugs, which are primarily functional bugs, can evade the detection of current tools. Automatically identifying functional bugs in smart contracts presents challenges from multiple perspectives. The primary issue is the significant gap between understanding the high-level logic of the business model and checking the low-level implementations in smart contracts. Furthermore, identifying deeply rooted functional bugs in smart contracts requires the automated generation of effective detection oracles based on various bug features.**

To address these challenges, we design and implement PROMFUZZ, an automated and scalable system to detect functional bugs in smart contracts. In PROMFUZZ, we first propose a novel Large Language Model (LLM)-driven analysis framework, which leverages a dual-agent prompt engineering strategy to pinpoint potentially vulnerable functions for further scrutiny. We then implement a dual-stage coupling approach, which focuses on generating invariant checkers that leverage logic information extracted from potentially vulnerable functions. Finally, we design a bug-oriented fuzzing engine, which maps the logical information from the high-level business model to the low-level smart contract implementations, and performs the bug-oriented fuzzing on targeted functions. We evaluate PROMFUZZ from 4 perspectives on 5 ground-truth datasets and compare it with multiple state-of-the-art methods. The results show that PROMFUZZ achieves 86.96% recall and 93.02% F1-score in detecting functional bugs, marking at least a 50% improvement in both metrics over state-of-the-art methods. Moreover, we perform an in-depth analysis on 10 real-world DeFi projects and detect 30 zero-day bugs. Our further case studies, the risky first deposit bug and the AMM price oracle manipulation bug on real-world DeFi projects, demonstrate the serious risks of the exploitable functional bugs in smart contracts. Up to now, 24 zero-day bugs have been assigned CVE IDs. Our discoveries have safeguarded assets totaling $18.2 billion from potential monetary losses.

I. INTRODUCTION

Blockchains have increasingly become a crucial component of the contemporary economic landscape. As of the writing of this paper, the total market capitalization of global cryptocurrencies has reached $3.5 trillion [3]. Smart contracts, which are fundamental pillars of the blockchain, facilitate the development of decentralized financial (DeFi) and various business transactions. Nevertheless, smart contracts are prone to exploitable bugs that often result in substantial financial losses. According to a report by CertiK, a leading Web3 security firm, over $1.8 billion was lost due to 751 security incidents in 2023 [1].

Currently, there are many works have focused on detecting exploitable bugs in smart contracts, which can be categorized into four main types based on their methodologies: static analysis [13], [38], fuzzing [10], [30], verification [32], [34], and symbolic execution [5], [31]. Despite these efforts, a recent study [45] reveals that over 80% of exploitable bugs remain undetected by current tools, which are primarily functional bugs. This limitation mainly stems from the tools' reliance on simple or hard-coded oracles to identify exploitable bugs. Nevertheless, detecting functional bugs involves comprehending the high-level business logic before scrutinizing the low-level implementations in smart contracts. Therefore, there is a pressing need for a practical system that can automatically detect functional bugs in smart contracts.

A. Challenges

To detect functional bugs in smart contracts automatically, we have the following key challenges.

Challenge I: Unraveling business logic in smart contracts. Detecting functional bugs in smart contracts requires a deep understanding of domain-specific properties intertwined with the contract's business logic. The first challenge lies in automating the accurate extraction of this high-level business logic, a critical step for enabling functional bug detection. Designing an effective method to achieve this goal is difficult since business logic is often complex and intricately embedded within the low-level code implementations of smart contracts, posing significant challenges even for seasoned human auditors in distinguishing between business and code logic.

✦: Xingshuang Lin and Qinge Xie are the co-first authors.
✉: Binbin Zhao is the corresponding author.

2643-1572/25/$31.00 ©2025 IEEE
DOI 10.1109/ASE63991.2025.00091

Challenge II: Bug checker generation. The business logic extracted from smart contracts is highly abstract and cannot be directly applied to functional bug detection. Thus, the second challenge involves deconstructing this business logic and creating practical bug checkers that can effectively utilize the extracted logic information. Designing bug checkers is non-trivial due to the varying fundamental characteristics exhibited by different types of functional bugs. Moreover, within diverse contract business contexts, the same type of functional bug may manifest different attributes, complicating the abstraction of functional bug features and the formulation of effective detection rules.

Challenge III: Functional bug detection. In practice, the effective deployment of bug checkers necessitates the support of sophisticated analysis tools for comprehensive bug detection. Currently, there is a gap in the availability of practical methods that can efficiently leverage these checkers. Therefore, the third challenge is to design a practical analysis method to efficiently utilize bug checkers for functional bug detection. Existing static analysis methods, while useful, struggle with false negatives as they cannot fully replicate the contract behavior in real-world environments, missing deep-level runtime-specific issues. On the other hand, traditional dynamic analysis methods explore programs randomly without a specific target, often relying on program crashes to identify bugs. However, functional bugs typically do not cause crashes and are unlikely to be triggered by random exploration alone.

B. Methodology

In this paper, we aim to address these challenges to detect functional bugs in smart contracts. To this end, we propose PROMFUZZ, an automated and practical system to conduct functional bug detection on smart contracts. Our design philosophy is as follows.

First, to solve the **Challenge I**, we start by designing a novel Large Language Model (LLM)-driven analysis framework. This framework is supported by a dual-agent prompt engineering strategy, featuring the Auditor Agent and the Attacker Agent. These specialized agents empower LLM to analyze smart contracts from two distinct perspectives: a meticulous auditor's viewpoint and a malicious attacker's perspective. We further enhance LLM's capabilities by developing a results fusion algorithm, combining insights from both the Attacker Agent and the Auditor Agent. This integration aids in pinpointing potentially vulnerable functions for detailed examination, significantly enhancing the robustness of LLM's output. **Second**, to address the **Challenge II**, we propose a dual-stage coupling approach, which focuses on generating invariant checkers that leverage logic information extracted from potentially vulnerable functions. In the initial stage, we implement a hierarchical matching approach that utilizes LLM's capabilities to identify critical variables and principal statements relevant to various bug features. Subsequently, in the second stage, we implement a template-based checker generation method. This method involves designing 6 invariant checker templates capable of processing critical variables and principal statements to produce final invariant checkers. This dual-stage approach mitigates the randomness and hallucination inherent in LLM by breaking down complex tasks into simpler components, ensuring more reliable invariant checker generation. **Third**, to solve the **Challenge III**, we implement a bug-oriented analysis engine. The main idea behind our design is to strategically insert invariant checkers into specific potentially vulnerable functions, guiding the engine to explore these critical areas rather than searching aimlessly. By integrating invariant checkers into our analysis engine, we successfully detect functional bugs in smart contracts with high recall and F1-score.

C. Contributions

We summarize our main contributions as follows.

- We propose PROMFUZZ, an automated and practical system to detect exploitable functional bugs in smart contracts, which fills the gap between understanding the high-level business logic and scrutinizing the low-level implementations in smart contracts. We employ a novel dual-agent prompt engineering strategy that enables LLM to effectively extract hidden business logic from smart contracts. Additionally, our bug-oriented fuzzing engine successfully maps the logical information from the high-level business model to the low-level smart contract implementations.
- To the best of our knowledge, we build the first ground-truth dataset for verified functional bugs in smart contracts, including 261 contracts from various DeFi projects. To facilitate future blockchain security research, we open-source both PROMFUZZ and the dataset at https://github.com/promfuzz.
- We have implemented and evaluated PROMFUZZ across 4 dimensions using 5 ground-truth datasets. The results show that PROMFUZZ achieves 86.96% recall and 93.02% F1-score in detecting functional bugs. PROMFUZZ demonstrates its superiority in at least a 50% improvement in both metrics over state-of-the-art methods.
- Our extensive analysis of 10 real-world DeFi projects demonstrates PROMFUZZ's great performance in detecting real-world functional bugs. Up to now, PROMFUZZ has identified 30 zero-day bugs in 6 of the 10 real-world DeFi projects, with 24 of these bugs have been assigned CVE IDs. Our discoveries have safeguarded assets totaling $18.2 billion from potential monetary losses.

II. BACKGROUND

In this section, we offer a succinct overview of the key concepts and techniques utilized throughout the paper, along with a motivating example.

A. Functional Bugs in Smart Contract

In this paper, we primarily focus on four major categories of functional bugs, including a total of ten subcategories of functional bugs. These classifications cover the majority of functional bugs and are derived from prior research [33], [45].

Price Oracle Manipulation. In the blockchain ecosystem, smart contracts rely on price oracles to access real-world data

like cryptocurrency prices. However, these oracles are susceptible to manipulation attacks, where attackers influence price data to impact transaction prices and gain illegal profits. We mainly focus on two specific price oracle manipulation bugs, Automated Market Maker (AMM) price oracle manipulation and non-AMM price oracle manipulation.

Unauthorized Behavior. Smart contracts often involve frequent token transfers, which, although a routine operation, are susceptible to potential attacks. These attacks stem from unauthorized behaviors—actions within a contract that deviate from its programmed rules or intended operational logic. Such bugs can result in loss of funds, corruption of the intended contract state, and other unintended effects that compromise the integrity and security of blockchain transactions. We specifically focus on two types of unauthorized behavior bugs: approval not clear and unauthorized transfer.

Insecure Calculating Logic. Smart contracts often exhibit vulnerabilities in their computational logic, which can lead to significant disruptions in their intended functions. A common issue arises when smart contracts do not accurately handle calculations related to share distributions, particularly during initial deposits. This can grant an unfair advantage to the first depositor, resulting in a disproportionate allocation of shares compared to subsequent participants. We focus on three types of insecure calculation logic bugs: wrong checkpoint order, wrong interest rate order, and risky first deposit.

Incorrect Control Mechanism. Smart contracts can suffer from critical flaws if they do not properly manage execution flows or restrict access to certain functionalities. Such mismanagement can trigger unintended behaviors or create vulnerabilities that attackers can exploit. This paper delves into three specific types of incorrect control mechanism bugs: improper handling of the deposit fee, wrong implementation of amount lock, and vote manipulation.

B. Large Language Models

Large Language Models (LLMs) have demonstrated significant advancements across a variety of tasks, including code generation [46], static analysis [20], and program repair [42]. Currently, a set of prominent LLMs have been developed, such as the Generative Pre-trained Transformer (GPT) by OpenAI, LLaMA [35] by Meta, and Gemini [7] by Google. These models possess extensive world knowledge, strong problem-solving capabilities, sophisticated reasoning skills, and a robust capacity for instruction following.

LLMs exhibit significant emergent abilities [40] that set them apart from traditional pre-trained models. They have three notable emergent abilities: in-context learning [9], instruction following [27], and step-by-step reasoning [29]. For instance, small language models often struggle with complex tasks that require multi-step reasoning, such as analyzing the business logic concealed within smart contracts. In contrast, LLMs can adeptly manage these challenges using their emergent abilities. They utilize techniques like Chain-of-Thought (CoT) [41] prompting to improve inference, thereby enabling them to address complex challenges effectively. In this paper, we utilize **GPT-4-turbo** as the foundational model in PROMFUZZ to analyze the hidden business logic in smart contracts. We focus on designing novel prompt strategies to trigger its emergent capabilities and implement practical measures to enhance its output robustness.

C. Motivating Example

We provide a functional bug discovered on Code4rena [2] as a motivating example, as shown in Figure 1. In the SimplePool contract, the function transferFrom is designed to transfer to the user the share tokens purchased with base tokens, while simultaneously deducting the corresponding amount from the user's base token allowance. The auxiliary function balanceToShares converts a specified quantity of base tokens into share tokens according to the parameter pricePerShare, which defines the exchange rate between base tokens and share tokens. However, a critical flaw emerges from an incorrect subtraction operation at line 9 that allows users to acquire share tokens at an unfairly low cost. For instance, when pricePerShare = $1e18$, a user providing $5e18$ base tokens to the function balanceToShares at line 7 will receive 5 share tokens. These share tokens are then transferred through the function _transfer at line 8. While the conversion and transfer steps are correctly executed, the functional bug manifests in the subsequent deduction of payment. At line 9, the implementation erroneously deducts only 5 base tokens because the subtraction is applied to the variable amountInShare (equal to 5), rather than the correct variable amount (equal to $5e18$). The correct implementation should be updated as follows: _approve(sender,_msgSender(),_allowances[sender][_msgSender()].sub(amount,"ERC20: transfer amount exceeds allowance"));. As a result, the user obtains 5 share tokens for only 5 base tokens, a price significantly below the intended market rate. By repeatedly invoking transferFrom, a user can continue to exploit this functional bug to acquire additional share tokens at an unfairly low cost.

```
1  contract SimplePool{
2    // In this contract, we omit irrelevant code.
3    uint256 public pricePerShare;
4    function balanceToShares(uint256 balance) public view
         returns (uint256) {
5      return balance.mul(1e18).div(pricePerShare); }
6    function transferFrom(address sender, address
         recipient, uint256 amount) public virtual
         override returns (bool) {
7      uint256 amountInShares = balanceToShares(amount);
8      _transfer(sender, recipient, amountInShares);
9      _approve(sender, _msgSender(), _allowances[sender][
           _msgSender()].sub(amountInShares, "ERC20:
           transfer amount exceeds allowance"));
10     return true; }}
```

Fig. 1: The Approval Not Clear bug (line 9).

A series of works, such as *GPTScan* [33] and *SMARTINV* [37], have explored the use of LLMs to detect functional bugs in smart contracts. *GPTScan* employs GPT-based matching analysis to initially identify potential functional bugs, which are then confirmed through static analysis. *SMARTINV*

utilizes a fine-tuned LLaMa-7B model to infer invariants relevant to functional bug analysis and confirm bugs using validation algorithms. However, they are not consistently effective in identifying such bugs. Specifically, *GPTScan* cannot detect this issue as it fails during the GPT-based matching phase. *SMARTINV* generates invariants such as pricePerShare > 0 and balanceToShares(amount) > 0, which fail to capture the necessary details for analyzing this bug effectively.

PROMFUZZ introduces a novel approach that could effectively tackle and detect such bugs. Initially, PROMFUZZ employs an LLM-driven multi-perspective analysis that uses iterative questioning to pinpoint potential occurrences of this bug. Following this identification, PROMFUZZ extracts critical variables, including sender, _allowances, and amount, to construct the invariant checker. The core idea for detecting the functional bug in the motivating example is to verify whether the user's base token allowance before the payment, minus the required payment amount, equals the allowance after the payment. Specifically, the invariant checker records the user's base token allowance before line 9 as old_allowance and inserts the invariant assertion, assert (old_allowance − amount == _allowances[sender][_msgSender()]);, immediately following line 9. Finally, PROMFUZZ involves bug-oriented fuzzing to ensure a comprehensive identification of functional bugs, completing the detection process.

III. PROMFUZZ DESIGN

In this section, we present the design details of PROMFUZZ. At a high level, PROMFUZZ aims to automatically find functional bugs in smart contracts. As shown in Figure 2, PROMFUZZ mainly consists of four modules. First, the LLM-driven multi-perspective analysis module accepts smart contracts as input and utilizes a dual-agent prompt engineering strategy, featuring the Auditor Agent and the Attacker Agent. These agents enable GPT to analyze smart contracts from distinct perspectives: a meticulous auditor's viewpoint and a malicious attacker's perspective, aiding in pinpointing potentially vulnerable functions accurately. Next, the invariant checker generation module leverages a dual-stage coupling approach, which focuses on generating invariant checkers that leverage logic information extracted from potentially vulnerable functions. Then, the bug-oriented analysis engine module strategically inserts the invariant checkers into specific potentially vulnerable functions, guiding the engine to explore these critical areas rather than searching aimlessly. Finally, the functional bug detection module executes comprehensive bug detection on smart contracts.

A. LLM-driven Multi-Perspective Analysis

The purpose of LLM-driven multi-perspective analysis is to analyze the high-level business logic concealed within smart contracts and pinpoint potentially vulnerable functions.

Detecting functional bugs in smart contracts requires a comprehensive understanding of the high-level business logic. Nevertheless, it is challenging to design a practical system that can automatically analyze the obscured business logic

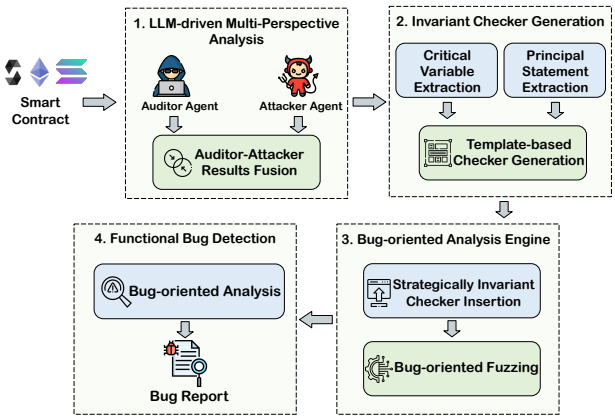

Fig. 2: Framework of PROMFUZZ.

of smart contracts. The main problem lies in the fact that the high-level business logic is obscured within the low-level implementations of smart contracts, making it inaccessible through simple or hand-coded oracles that are commonly adopted by previous works.

To address the above problem, we propose a novel LLM-driven analysis framework that employs a dual-agent prompt engineering strategy. This framework incorporates two specialized agents: the Auditor Agent and the Attacker Agent, each designed to enhance the emergent capabilities of GPT in different ways. Specifically, we regard the process of identifying bugs in smart contracts as an auditing task. Thus, the Auditor Agent is designed to act as a meticulous auditor, scanning smart contracts thoroughly and then detecting potential bugs. However, relying solely on the Auditor Agent might result in false positives and potentially overlook critical vulnerabilities, which could lead to irreversible damages if not identified early. To address this limitation, we introduce the Attacker Agent. This agent analyzes smart contract code from an attacker's perspective, aiming to uncover exploitable weaknesses that the Auditor Agent might miss. This novel dual-agent architecture enables precise identification and reliable assessment of potential bugs, thereby enhancing the robustness of GPT's output.

In our dual-agent framework, the Auditor Agent and the Attacker Agent adopt distinct analysis strategies that differ in granularity. The intuition behind our design is that the two agents serve fundamentally different roles in the analysis pipeline. The Auditor Agent performs fine-grained inspections, dissecting the code at a detailed level to examine its semantics, business scenarios, and logical correctness. This mirrors real-world auditing practices, in which security professionals meticulously classify and validate individual program components to ensure reliability and security. Based on this fine-grained analysis, the Auditor Agent produces subcategories of bugs, offering a nuanced characterization of potential functional bugs. In contrast, the Attacker Agent conducts a coarse-grained analysis, focusing less on the accuracy of code understanding and more on the broader issue of exploitability. Rather than scrutinizing every line of code, the Attacker Agent

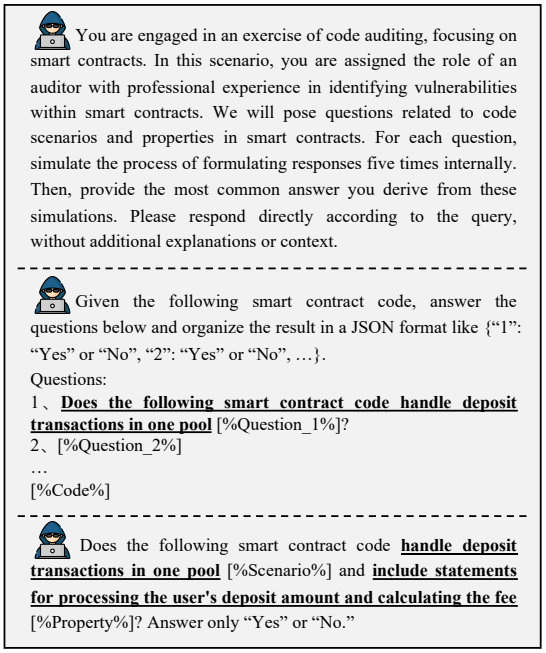

Fig. 3: Prompt template for the Auditor Agent.

seeks to determine whether certain functions, state transitions, or contract behaviors can be exploited to achieve unintended outcomes. Based on this coarse-grained analysis, the Attacker Agent outputs the primary categories of bugs, emphasizing the overarching attack surfaces that are most relevant for exploitation. This difference in analytical granularity reflects the agents' fundamentally different perspectives on functional bug assessment: the Auditor Agent prioritizes accuracy and comprehensive understanding, while the Attacker Agent prioritizes feasibility of exploitation and its potential impact.

Both agents are equipped with precisely crafted prompts that aid in analyzing the obscured business logic of smart contracts. This dual-agent strategy ensures a thorough evaluation by encompassing both defensive and offensive aspects of security. By synthesizing the insights from both the Auditor Agent and the Attacker Agent, our framework significantly improves the precision of the analysis. It minimizes the removal of false positives while ensuring that critical bugs are accurately identified and retained for further action. In the following, we delve into the details of the dual-agent prompt engineering strategy employed in PROMFUZZ.

Auditor Agent Design. In the real world, smart contract auditors primarily concentrate on securing the core code and business operations of smart contracts. Inspired by this practical approach, we have designed our Auditor Agent to mirror the real-world auditing process. However, crafting the Auditor Agent is not trivial, as it involves a two-step process: initially capturing the essential code or business operations of smart contracts and subsequently identifying any abnormal behaviors within these core elements.

To address the above challenges, we design the Auditor Agent based on the CoT prompting strategy and the scenario and property matching methodology proposed by *GPTScan* [33]. A "scenario" describes the code functionality under which a functional bug might occur, while a "property" details the vulnerable code attributes or operations. *GPTScan* initially selects ten representative functional bugs in smart contracts, following the classifications by Zhang et al. [45]. Upon further analysis, we observe that these categories could be streamlined for greater clarity and applicability.

Our refined approach consolidates the initial ten categories into four primary categories of functional bugs. Notably, we merge the categories of *price manipulation by buying tokens* and *slippage* into a single, more comprehensive category named *non-AMM price oracle manipulation*. Moreover, our manual analysis of the functional bugs leads to the identification of two new subcategories: *improper handling of the deposit fee* and *wrong implementation of amount lock*.

Figure 3 outlines the prompt template for the Auditor Agent. First, GPT is informed that it is participating in a code auditing exercise. We then allow GPT to internally simulate the response formulation process five times and provide the most common answer to mitigate the randomness of the responses. Next, we input the target smart contract code and pose questions related to the functionality under which a functional bug might occur—i.e., the scenario. This question aids in identifying potential bug categories within the code. Finally, for any scenario where GPT confirms a potential issue, we follow up with specific questions about the vulnerable code attributes or operations—i.e., the property. If GPT responds affirmatively, we consider the presence of a functional bug; if not, we rule out that specific bug. Designing prompts requires only a one-time manual effort and can be easily extended to cover most functional bugs.

Attacker Agent Design. The Auditor Agent, although structured with a step-by-step reasoning process following conventional audit procedures, may lead to false positives and overlook critical bugs if solely relied upon. To address these limitations, we propose the Attacker Agent, which is specifically designed to emulate the cognitive processes of an attacker, actively searching for and exploiting bugs within smart contracts. By identifying and leveraging these weak points, the Attacker Agent enables GPT to conduct smart contract audits from an adversarial viewpoint. Designing the Attacker Agent poses significant challenges, as it requires a precise understanding and summarization of the manifestations and defensive measures associated with common logical vulnerabilities in smart contracts.

To address the above challenges, we gather an extensive dataset of real-world attack incidents for each bug category. Through detailed manual analysis of these incidents, we extract critical information, e.g., distinctive bug features, to craft effective prompts based on the CoT prompt strategy. Bug features refer to attributes or behaviors in the code that present security risks under certain conditions, which could potentially be exploited under specific conditions to carry out malicious activities. The process of utilizing the Attacker Agent involves several key steps, outlined in Figure 4. First, similar to the

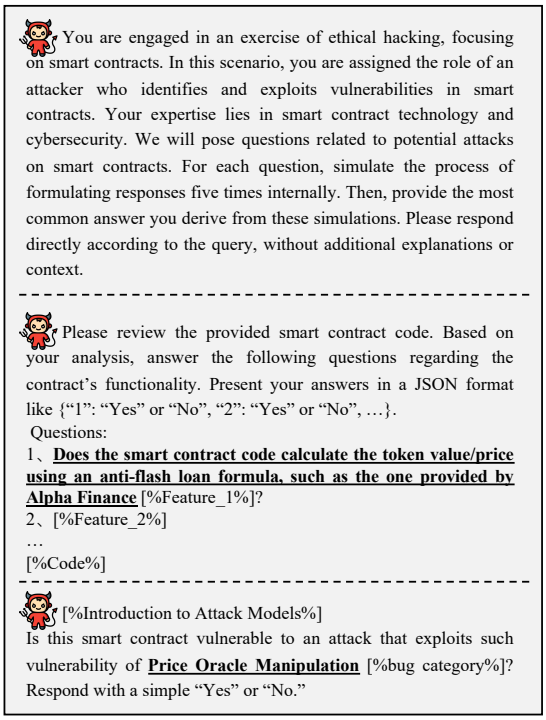

Fig. 4: Prompt template for the Attacker Agent.

Auditor Agent, GPT is informed that it is participating in an ethical hacking exercise. Next, the target smart contract code is inputted, and GPT is questioned about potential bug features that could be exploited for malicious activities. Finally, for bugs whose features have been recognized during the last question, we provide the corresponding attack model for these bugs as prompts to GPT. We then query GPT whether the identified features could lead to exploitable attacks based on the given attack model. An affirmative response from GPT indicates the presence of a functional bug, while a negative response helps us dismiss that particular bug.

Auditor-Attacker Results Fusion. To derive more accurate classifications of bugs, we integrate the results from both the Auditor Agent and the Attacker Agent. This fusion approach leverages the coarse-grained primary category insights from the Attacker Agent to refine and enhance the fine-grained subcategory determinations made by the Auditor Agent. The Auditor-Attacker results fusion algorithm is provided in our supplementary material (§I).

B. Invariant Checker Generation

While we have identified potentially vulnerable functions, we cannot solely rely on GPT's results since its output includes randomness. To address this gap, it is crucial to develop a practical method that can minimize the impact of this randomness. One effective approach is to check the violation of invariants to further scrutinize the identified bugs. In smart contracts, an invariant is a condition or a set of conditions that always hold true regardless of the state of the contract at any point in its execution. These invariants are crucial for ensuring that the contract behaves correctly and securely, safeguarding against unintended actions and vulnerabilities. Therefore, there is a pressing need to implement a robust method for generating invariants to enhance functional bug detection. The primary difficulty in generating valid invariants lies in tailoring them specifically to different types of bugs. This requires a deep understanding of both the specific features of each bug and the context of the vulnerable code snippet.

To address these problems, we design a dual-stage coupling method to generate invariant checkers that take logic information extracted from potentially vulnerable functions as input and generate corresponding invariants. Specifically, in the first stage, we implement a hierarchical matching approach that utilizes GPT's capabilities to extract essential variables and statements. This preliminary step focuses on gathering critical information rather than generating complete invariants directly. In the second stage, we introduce a template-based checker generation approach. We design 6 invariant checker templates specifically tailored to integrate critical variables and principal statements extracted in the first stage. This coupling ensures the generated invariants are both relevant and robust. In the following, we present the details of the invariant checker generation.

Critical Variables and Principal Statements Extraction. To develop the invariant checker, it is essential to initially identify and extract critical variables and principal statements related to the business logic from smart contracts. These elements are important in determining the invariants associated with different types of bugs, which are then used to populate a predefined checker template. However, the identification of these critical elements poses significant challenges. It requires a thorough understanding of the invariants pertinent to various types of bugs and the design of a practical and effective method for their extraction from smart contracts.

In response to this challenge, we propose a hierarchical matching method that capitalizes on the capabilities of GPT for the extraction of critical variables and principal statements. Our approach unfolds in two primary steps. First, we perform an in-depth analysis of each bug category to pinpoint critical variables and principal statements that are vital for comprehending their logic. For instance, in the scenario of an AMM price oracle manipulation bug, a critical variable is the one that holds the calculated price of the LP token. We present the characteristics of critical variables and principal statements for each bug subcategory in our supplementary material(§II). Second, leveraging the identified elements, we formulate specific prompts and input them into GPT to query each vulnerable function as identified by the LLM-driven multi-perspective analysis module. Figure 5 illustrates the prompt template designed to extract critical variables and principal statements from code snippets potentially vulnerable to AMM price oracle manipulation. For different bug categories, we can substitute [%Critical Variable%] and [%Principal Statement%] in the template with the predefined elements specific to those bugs. With these tailored prompts, GPT is able to precisely pinpoint the relevant critical variables and principal statements,

enhancing the accuracy of our bug detection process.

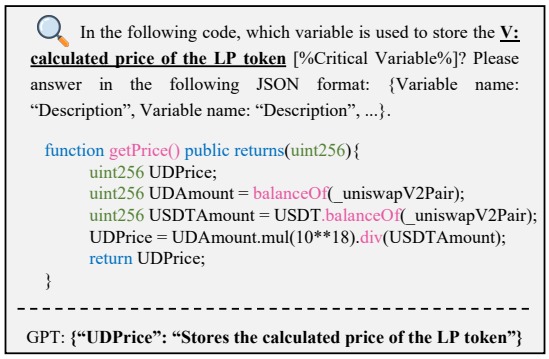

Fig. 5: Prompt template for extracting critical variables from code snippets potentially vulnerable to AMM price oracle manipulation.

Template-based Checker Generation. While we have extracted critical variables and principal statements from smart contracts, we currently lack an efficient method to convert these into usable invariants for bug detection. To fill this gap, we propose a template-based checker generation method. Specifically, we mainly design 6 types of checkers: **PriceChange_Checker**, **ExchangeRate_Checker**, **TokenChange_Checker**, **StatementOrder_Checker**, **ShareSafety_Checker**, and **StateChange_Checker**. For a detailed invariant checker templates, please see our supplementary material(§III). Each checker includes the conditional statements that assesses the contract state. For instance, as shown in Figure 6, the **PriceChange_Checker** for AMM price oracle manipulation determines whether the variable (i.e., calculated price of the LP token) falls below 90% or exceeds 110% of the old price. Such a condition indicates that the price change surpasses the ±10% threshold, typically signaling abnormal price fluctuations and triggering an alert for an existing AMM price oracle manipulation bug. These checkers could be dynamically combined to comprehensively cover a broad range of functional bugs, ensuring scalability.

```
uint256 oldpricde = Initial Price
.....
assert (100 * Variable <= 90 * oldprice || 100 * Variable >= 110 * oldprice);
oldprice = Variable;
```

Fig. 6: The checker for AMM price oracle manipulation.

C. Bug-Oriented Analysis Engine

Though we have identified potentially vulnerable functions in smart contracts and generated the corresponding invariant checkers, we still lack a practical method to leverage them for functional bug detection. To tackle the above problem, we implement a bug-oriented analysis engine with the following designs.

Invariant Checker Insertion. Inserting invariant checkers into smart contracts requires careful consideration to avoid false negatives caused by incorrect or inappropriate insertion points. We propose two criteria for strategic insertion: the variable-oriented invariant insertion criterion and the statement-oriented invariant insertion criterion. The variable-oriented criterion focuses on inserting invariants before and after critical code blocks that manipulate critical variables, ensuring that the operations conform to the invariants. Similarly, the statement-oriented criterion places invariants around crucial statements to verify if their execution logic and sequence adhere to the defined invariants. Utilizing these criteria, invariant checkers are automatically positioned in optimal locations within the code, taking into account the logical relationships between critical variables, principal statements, and their positions in the code block.

Algorithm 1: Functional Bug Detection

Input: C: smart contract code
B: potential functional bug subcategory
Output: R: the set of detected functional bugs

1: **for** each $B_i \in B$ **do**
2: $v, s = Extract\_Variables\_and\_Statements(B_i, C)$
3: $inv = Generating\_invariants(v, s)$
4: $CwI = Invariant\_Checker\_Insertion(inv, B_i, C)$
5: $C_s \leftarrow initial\_infant\_state\_corpus$
6: $C_{ts} \leftarrow initial\_transaction\_and\_state\_pair\_corpus$
7: **while** $t \neq \varnothing$ **do**
8: $s_{mut}, t_{mut} \leftarrow input\_mutation(C_{ts}, C_s)$
9: $f, s', violation \leftarrow Execution(s_{mut}, t_{mut}, CwI)$
10: **if** $violation\ is\ found$ **then**
11: $R = R.append(inv, B_i)$
12: **end if**
13: $update\_corpus(f, s', s_{mut}, t_{mut})$
14: **end while**
15: **end for**
16: **return** R

Bug-Oriented Fuzzing. After inserting invariant checkers into smart contracts, we then employ fuzzing to address the aforementioned limitations in static analysis methods. Traditional dynamic analysis techniques often explore programs randomly without a specific target, typically identifying bugs through program crashes. However, functional bugs in smart contracts do not generally lead to crashes and are less likely to be detected through random testing. To mitigate this, we implement a bug-oriented fuzzing approach that directs the fuzzer to specifically target functions with inserted invariant checkers, rather than performing random exploration. Our method involves inserting a specific bug alert function into each invariant checker, which allows us to direct the fuzzer to activate these targeted checkers rather than engaging in random exploration. Our method significantly improves the efficiency and effectiveness of the fuzzing process.

D. Functional Bug Detection

The functional bug detection module aims to perform comprehensive bug detection on smart contracts. Algorithm 1 outlines the procedure for applying this methodology. Specifically, in this module, we first compile the target smart contracts with strategically inserted invariant checkers (line 2-4). Next, we perform fuzzing on these smart contracts (line 5-14). When the

fuzzer activates an invariant checker and detects a discrepancy, it signals the presence of a bug (line 10-12). According to the results, we then generate a comprehensive bug report that documents all identified functional bugs within the smart contracts.

IV. System Evaluation

In this section, we evaluate the performance and capabilities of PromFuzz from multiple perspectives. Our evaluation is structured around the following key research questions:

RQ1: How does the dual-agent prompt engineering strategy contribute to the performance of PromFuzz? (Section IV-B)

RQ2: What is PromFuzz's accuracy in generating invariant checkers? (Section IV-C)

RQ3: How effective is PromFuzz at detecting functional bugs in smart contracts? (Section IV-D)

RQ4: Is PromFuzz capable of discovering zero-day functional bugs in real-world smart contracts? (Section IV-E)

A. Dataset

To ensure a comprehensive assessment of PromFuzz's effectiveness, we systematically collect 459 smart contracts from various sources, including DeFiHackLabs [6], Web3bugs [45], SmartBugs [12], VeriSmart [32], on-chain smart contracts, and real-world DeFi projects. After further analysis, we remove 198 contracts that either lack functional bugs or are missing essential external files or configuration information required for compilation. Ultimately, we create 5 datasets comprising 261 contracts, covering a broad spectrum of functional bug categories and evaluation perspectives. A detailed description of these datasets is provided in Table I.

TABLE I: The evaluation datasets.

Dataset	Description	RQ
$D_{Exploit-SC}$	7 real-world attack cases with 47 contracts	RQ1,RQ3
D_{BV-SC}	10 real-world audit cases with 36 contracts	RQ1,RQ3
$D_{Synth-SC}$	6 real-world synthetic cases with 44 contracts	RQ1,RQ3
D_{SE-SC}	26 contracts with 59 extracted symbols	RQ2
$D_{Real-DeFi}$	10 real-world DeFi projects with 108 contracts	RQ4

Exploited smart contract dataset ($D_{Exploit-SC}$) includes smart contracts that have been subjected to real-world attacks in the past, each causing significant monetary losses. $D_{Exploit-SC}$ includes 7 real-world attack cases, associated with 47 contracts, all of which we have manually verified to be caused by functional bugs.

Bug-verified smart contract dataset (D_{BV-SC}) includes smart contracts audited through platforms like Code4rena [2] and Immunefi [4]. D_{BV-SC} includes 10 real-world audit cases, associated with 36 contracts, all of which have been confirmed by the developers to contain high-risk functional bugs.

Synthetic smart contract dataset ($D_{Synth-SC}$) consists of smart contracts where functional issues have been deliberately introduced. The purpose of this dataset is to expand $D_{Exploit-SC}$ and D_{BV-SC}, enabling a more comprehensive evaluation of PromFuzz. $D_{Synth-SC}$ includes 6 synthetic cases across 44 smart contracts. We construct $D_{Synth-SC}$ with the following steps. First, we collect smart contracts that are well-designed, well-audited, and widely deployed on mainstream blockchains, which reasonably indicates a low likelihood of major bugs. Then, two of our authors, who are experienced in smart contract development, perform controlled bug injection. One author implements the injection based on bug patterns summarized from real-world attack incidents, while the other conducts independent verification.

Symbol-extracted smart contract dataset (D_{SE-SC}) includes smart contract functions with manually extracted and verified critical variables and principal statements. Specifically, D_{SE-SC} includes 59 extracted critical variables and principal statements across 26 smart contract functions. In the construction of D_{SE-SC}, we assign two of our authors with expertise in smart contract security. The first author conducted a manual analysis of each smart contract, identifying critical variables and principal statements relevant to invariant checker generation and functional bug detection. The second author performed a secondary verification to ensure the accuracy and reliability of the extracted information.

Real-world DeFi project dataset ($D_{Real-DeFi}$) comprises DeFi projects currently undergoing public audits through various platforms. Specifically, $D_{Real-DeFi}$ includes 10 such DeFi projects with 108 smart contracts.

TABLE II: Comparison of Dual-agent and Auditor-agent.

Architecture	Overall				
	TP	FP	FN	Recall	F1-Score
Dual-agent	**21**	**34**	**2**	**91.30%**	**53.85%**
Auditor-agent	15	24	8	65.22%	48.39%

B. Dual-Agent Architecture Evaluation

This subsection addresses the effectiveness of the dual-agent architecture of PromFuzz. First, we conduct an ablation study to evaluate the performance differences between the Dual-agent based PromFuzz and a version that utilizes only the Auditor Agent. Second, we compare Dual-agent with two mainstream reasoning models: DeepSeek-R1 and OpenAI-o3. The above comparisons are conducted using three datasets: $D_{Exploit-SC}$, D_{BV-SC}, and $D_{Synth-SC}$.

Ablation Study. As shown in Table II, Dual-agent architecture shows substantial performance improvements over the Auditor-agent architecture. Specifically, Dual-agent architecture achieves a recall of 91.30% and an F1-score of 53.85%, identifying 21 true positives and 34 false positives across $D_{Exploit-SC}$, D_{BV-SC}, and $D_{Synth-SC}$ datasets, with only 2 false negatives. In contrast, Auditor-agent architecture detects fewer true positives and has more false negatives, achieving a recall of 65.22% and an F1-score of 48.39%.

By adopting our dual-agent prompt engineering strategy, Dual-agent architecture significantly improves true positives and eliminates false negatives. Although Dual-agent architecture exhibits a higher number of false positives, these are

manageable as they can be effectively filtered out during the subsequent fuzzing phase.

We further analyze the false positives and false negatives by Dual-agent architecture and Auditor-agent architecture. For Dual-agent architecture, though the Attacker Agent increases true positives by considering multiple angles, it still has limitations and can lead to over-generation of false positives. For Auditor-agent architecture, auditor's perspective, which focuses solely from an auditing standpoint, can result in underreporting, filtering out some high-risk vulnerabilities. The limitations of auditor prompt rules, which, like those of the attacker, cannot cover all bugs, resulting in some being missed. Besides, for both Dual-agent architecture and Auditor-agent architecture, GPT's understanding is still insufficient, leading to cases of prompt misinterpretation.

Comparison with reasoning models. As shown in Table III, Dual-agent architecture outperforms mainstream reasoning model architectures. Specifically, Dual-agent architecture ahieves a recall of 91.30% and an F1-score of 53.85%. OpenAI-o3 architecture achieves a recall of 52.17% and an F1-score of 32.43%. DeepSeek-R1 architecture achieves a recall of 47.83% and an F1-score of 39.29%.

We further analyze the false positives and false negatives by mainstream reasoning models. In OpenAI-o3 and DeepSeek-R1, the decomposition of the detection task is determined autonomously by the reasoning model. Such task planning can introduce a degree of randomness, potentially guiding the LLM toward incorrect reasoning paths and resulting in false positives and false negatives. In contrast, Dual-agent architecture designs its reasoning workflow based on strategies observed in real-world auditors and attackers, enabling the LLM to follow a more structured and reliable analysis process.

TABLE III: Comparison of Dual-agent and reasoning models.

Architecture	Overall				
	TP	FP	FN	Recall	F1-Score
Dual-agent	**21**	**34**	**2**	**91.30%**	**53.85%**
OpenAI-o3	12	39	11	52.17%	32.43%
DeepSeek-R1	11	22	12	47.83%	39.29%

C. Invariant Checker Generation Accuracy

This subsection answers RQ2. In this step, we mainly evaluate the accuracy of PROMFUZZ in extracting critical variables and principal statements from potentially vulnerable functions, which determines the accuracy of final invariant checkers. We deploy PROMFUZZ on dataset D_{SE-SC} for this evaluation.

As shown in Table IV, the invariant checker generation module demonstrates excellent performance in critical variable and principal statement extraction, achieving an overall precision of 94.92%. Specifically, for the variable extraction task, PROMFUZZ achieves a precision of 96.23%, with two false positive. The false positives are due to unconventional variable naming. For the statement extraction task, it achieves a precision of 83.33%, with only one false positive. This false positive arises from the complex business behaviors in smart contracts, where PROMFUZZ fails to differentiate between multiple similar behaviors in a single complex case. These critical variables and principal statements are expected to derive 26 correct invariant checkers but ultimately generate 24 correct ones, resulting in a precision of 92.31%.

TABLE IV: Variables and statements extraction accuracy.

Task	D_{SE-SC}		
	TP	FP	Precision
Variables Extraction	51	2	96.23%
Statements Extraction	5	1	83.33%
Overall	**56**	**3**	**94.92%**

D. Functional Bug Detection Accuracy

This subsection answers RQ3. In this step, we evaluate the functional bug detection accuracy of PROMFUZZ with two metrics: recall and F1-score. We compare PROMFUZZ with four off-the-shelf tools: *GPTScan*, *SMARTINV*, *ItyFuzz*, and *SMARTIAN* [10]. We also consider a comparison between PROMFUZZ and *PropertyGPT* [22]. However, due to the unavailability of its code, we are unable to conduct this comparison. We do not compare PROMFUZZ with several program verification-based techniques [18], [28], [11], since they are largely ineffective in detecting functional bugs, as their constraints are designed for implementation bugs. Both *ItyFuzz* and *SMARTIAN* are fuzzers designed for detecting bugs in smart contracts. We perform them on $D_{Exploit-SC}$, D_{BV-SC}, and $D_{Synth-SC}$.

TABLE V: Functional bug detection accuracy.

Tool	Overall				
	TP	FP	FN	Recall	F1-Score
PROMFUZZ	**20**	**0**	**3**	**86.96%**	**93.02%**
GPTScan	7	12	16	30.43%	33.33%
SMARTINV	6	43	17	26.09%	16.67%
ItyFuzz	0	0	23	0	0
SMARTIAN	3	0	16	15.79%	27.27%

As shown in Table V, PROMFUZZ demonstrates superior performance in detecting functional bugs across three datasets. PROMFUZZ achieves an overall recall of 86.96% and an F1-score of 93.02%, detecting 20 true positives and maintaining 0 false positives across all datasets, with only 3 false negatives. Further analysis of the false negatives identified by PROMFUZZ reveals two primary causes. First, one false negative results from inaccuracies in the extraction of critical variables and principal statements. Second, the remaining two false negatives occur during the LLM-driven analysis step, where GPT fails to identify two critical bugs. Besides, with each smart contract project scan, PROMFUZZ takes an average of 43.8 seconds and costs approximately $0.15, demonstrating its great efficiency and cost-effectiveness.

In comparison, *GPTScan* and *SMARTINV* yield similar numbers of true positives and false negatives. However,

TABLE VI: Zero-day functional bug detection results. **TVL** represents the total value of cryptocurrency assets locked in the respective DeFi projects.

DeFi Project	Project Type	TVL	# Zero-day Bug	# CVE
1	Yield & Lengding	$ 136.8M	7	7
2	Yield	$ 14,300M	3	3
3	Yield	$ 0.172M	4	4
4	Exchanges	$ 102M	5	5
5	Yield & Sericves	$ 3,700M	5	5
6	Exchanges	$ 1.79M	6	0
Total		$ 18.2B	30	24

SMARTINV records the highest number of false positives among the five tools assessed. Furthermore, *ItyFuzz* fails to identify functional bugs without the specific invariants we provide. *SMARTIAN*, another fuzzing tool, shows inconsistent performance as it fails to operate on several smart contract files. Consequently, we exclude *SMARTIAN* from testing against all bugs. We further analyze the false positives and false negatives of these tools. . For *GPTScan*, the primary reason arises from its single-agent design, which can lead to the incorrect identification of correct behaviors as functional bugs or the failure to detect genuine flaws. For *SMARTINV*, the primary reason arises from its lack of comprehensive reasoning capabilities, resulting in some bugs being overlooked or misidentified. Both *ItyFuzz* and *SMARTIAN* are limited by their original oracles, which lack the capability to identify functional bugs effectively.

E. Zero-day Functional Bug Detection

This subsection answers RQ4. We perform PROMFUZZ on $D_{Real-DeFi}$ to evaluate its performance in detecting zero-day functional bugs in real-world smart contracts. As illustrated in Table VI, PROMFUZZ successfully identifies 30 zero-day functional bugs across 6 of the 10 Defi project. We have reported all detected bugs to the respective vendors, and 24 of these have been assigned CVE IDs. The total market value of DeFi projects affected by these bugs is estimated at $18.2 billion. As these bugs have not yet been fully resolved, we maintain confidentiality by anonymizing the details of both the vendors and the specific CVE information. Moreover, we notice that zero-day bugs primarily fall into four categories: 13 bugs involve AMM price oracle manipulation, 7 concern non-AMM price oracle manipulation, 4 are related to wrong checkpoint order, 4 pertain to risky first deposit, and 2 involve wrong interest rate order. Further analysis in Section V delves into the practical impacts of these bugs, examining specific cases to illustrate how attackers exploit these bugs.

V. CASE STUDY

To obtain an in-depth understanding of the practical impacts of functional bugs, we detail two such bugs identified by PROMFUZZ in real-world DeFi projects.

Risky First Deposit. As shown in Figure 7, the code contains a risky first deposit bug that can lead to an unfair advantage for the first user and a distorted distribution of

```
1  contract SimplePool{
2    function _deposit(address _recipient, uint256 _amount)
         internal returns (uint256) {
3      totalValueInLp += _amount;
4      uint256 share = _sharesForDepositAmount(_amount);
5      if (_amount > type(uint128).max || _amount == 0 ||
           share == 0){ revert InvalidAmount(); }
6      eETH.mintShares(_recipient, share);
7      return share;  }
8    function _sharesForDepositAmount(uint256
         _depositAmount) internal view returns (uint256) {
9      uint256 totalPooledEther = getTotalPooledEther() -
           _depositAmount;
10     if (totalPooledEther == 0){ return _depositAmount; }
11     return (_depositAmount * eETH.totalShares()) /
           totalPooledEther; } }
```

Fig. 7: The Risky First Deposit bug (line 10).

shares. Specifically, at line 10, when totalPooledEther == 0, the share variable at line 4 is set equal to _depositAmount, since the return value is assigned to _depositAmount in the _sharesForDepositAmount() function. Subsequent depositors have their shares calculated by (_depositAmount × eETH.totalShares()) / totalPooledEther, which depends on the values of eETH.totalShares() and totalPooledEther. While the calculation logic seems standard for compliant depositors, the first depositor can exploit the system by manipulating totalPooledEther. We can front-run other depositors' transactions and inflate the price of pool tokens through a substantial "donation." For instance, a malicious early user could deposit 1 wei of asset tokens as the first depositor, receiving 1 wei of shares tokens. Subsequently, the malicious sends 1 ETH to the pool, inflating totalPooledEther to 1 ETH + 1 wei (equal to $(1e18 + 1)$ wei). In this scenario, the second depositor, depositing 2 ETH (equal to $2e18$ wei) of asset tokens, would receive only 1 wei of shares tokens. Because 1.99 wei of shares tokens (calculated by $2e18/(1e18+1)$) will be rounded down to 1 wei of shares tokens in smart contracts. They will lose 0.5 ETH if they withdraw right after the _deposit(). This disproportionate calculation of the first user's shares can disadvantage subsequent users who receive fewer shares relative to their deposits.

AMM Price Oracle Manipulation. As shown in Figure 8, the code contains an AMM price oracle manipulation bug, which arises from the method of calculating token prices based solely on the supplies of two tokens. Specifically, _reserveA and _reserveB represent the respective reserve amounts of two assets. In line 13 of the code, the amount of token B is calculated using the amount of token A, along with the reserves of token A and token B. However, if the reserve amount of either token is maliciously manipulated, it can distort the price and lead to significant price fluctuations. For instance, consider token exchanges in the smart contract using the PoolHelpers library. Initially, the pool contains 500 token A and 500 token B. An attacker exchanges 100 token A for 100 token B, changing the pool to 600 token A and 400 token B. Then, the second user exchanges 30 token A and can only get 20 token B in return, leaving the pool with 630 token A and 380 token B. Finally, the attacker uses the 100 token B

```
1  library PoolHelpers {
2    function quote(uint256 _amountA, uint256 _reserveA,
         uint256 _reserveB) internal pure returns (uint256
          amountB_) {
3      require(_amountA != 0, "SmardexHelper:
           INSUFFICIENT_AMOUNT");
4      require(_reserveA != 0 && _reserveB != 0, "
           SmardexHelper: INSUFFICIENT_LIQUIDITY");
5      amountB_ = (_amountA * _reserveB) / _reserveA; } }
6  contract SimplePool{
7    address tokenA,tokenB;
8    function swap(uint256 _amountA,address _tokenA) return
         (uint256 _amountB){
9      require(_tokenA==tokenA||_tokenA==tokenB);
10     address _tokenB = (_tokenA==tokenA)?tokenB:tokenA;
11     _reserveA = getReservers(_tokenA);
12     _reserveB = getReservers(_tokenB);
13     _amountB = PoolHelpers.quote(_amountA, _reserveA,
           _reserveB); } }
```

Fig. 8: The AMM Price Oracle Manipulation bug (line 13).

they obtained to exchange for 165 token A, ultimately making a profit of 65 token A.

VI. Discussion

While PromFuzz achieves significant results in detecting functional bugs in smart contracts, it has several limitations. First, some false negatives occur due to inaccuracies in extracting critical variables and the inherent randomness in the LLM-driven analysis. Despite these issues, PromFuzz still outperforms existing methods in uncovering deeper functional bugs. Additionally, our bug-oriented analysis currently targets four main categories with ten subcategories, potentially missing some complex bug types. In the future, we plan to fine-tune a specific LLM for enhanced detection of functional bugs in smart contracts and develop new prompts to cover a wider range of bug categories, thus improving PromFuzz's comprehensiveness. Moreover, since our analysis partially relies on *ItyFuzz*, which is not specifically designed for functional bugs and lacks full utilization of contract status information, future efforts will focus on better integrating LLMs with dynamic analysis techniques to more effectively and comprehensively utilize contract status information, aiming to capture deeper functional bugs.

VII. Related Work

Recent efforts to evaluate the security of smart contracts have produced various methods, categorized into static analysis, dynamic analysis, and AI-based analysis.

Static Analysis. Tools like *Slither* [13], *Zeus* [18], *Securify* [36], *Madmax* [14], *Invcon* [21], *Verismart* [32], and *Verisol* [39] analyze smart contract code statically. *Slither* converts Solidity code into SlithIR to find bugs. *Securify* uses compliance and violation patterns to assess code safety. *Madmax* decompiles EVM bytecode and uses a logic-based specification for bug detection. *Verisol* translates Solidity into Boogie, formalizing semantic conformance against state machine workflows and reducing semantic checking to safety verification. However, these tools primarily focus on implementation bugs like integer overflow and reentrancy, and are less effective at identifying functional bugs.

Dynamic Analysis. Tools like *ItyFuzz* [30], *Smartian* [10], *Harvey* [43], *Echidna* [15], *ContractFuzzer* [17], *Mythril* [5], *Sailfish* [8], *Manticore* [26], *Teether* [19], and *Oyente* [25] analyzes programs during execution. *ItyFuzz* uses snapshot-based fuzzing to reduce re-execution overhead, prioritizing intriguing snapshot states. *Smartian* combines static analysis for seed generation with feedback mechanisms. *Sailfish* identifies state-inconsistency bugs using a combination of lightweight exploration and symbolic evaluation. While dynamic analysis captures more realistic behaviors by executing smart contracts, it still primarily targets implementation bugs and struggles to effectively detect functional bugs.

AI-based Analysis. *SmarTest* [31], *ILF* [16],*xFuzz* [44], and *GNN-based tools* [47], [23], [24] use AI to enhance smart contract bug detection. *SmarTest* combines symbolic execution with a language model to prioritize bugs. *GNN-based tools* leverage graph neural networks with expert patterns. While these tools improve detection capabilities, they mainly address implementation bugs. Recently, LLM-based tools like *GPTScan* [33] and *SMARTINV* [37] have been used to address functional bugs by analyzing discrepancies between business logic and code. These works, including the PromFuzz introduced here, open new avenues for detecting functional bugs in smart contracts using AI's cognitive capabilities.

VIII. Conclusion

In this paper, we introduce PromFuzz, an automated and practical system designed to detect functional bugs in smart contracts, which bridges the gap between understanding the high-level business model logic and scrutinizing the low-level implementations in smart contracts. PromFuzz achieves 86.96% recall and 93.02% F1-score in detecting functional bugs, representing a significant performance improvement over state-of-the-art methods. Additionally, we have created the first ground-truth dataset for functional bugs including 261 contracts from various DeFi projects. Furthermore, our in-depth analysis of 10 real-world DeFi projects has identified 30 zero-day bugs, with 24 of them receiving CVE IDs. Our discoveries have safeguarded assets totaling $18.2 billion from potential monetary losses.

Acknowledgment

This work was partly supported by the NSFC under No. U244120033, U24A20336, 62172243, 62402425 and 62402418, the Open Research Fund of Engineering Research Center of Blockchain Application, Supervision And Management (Southeast University), Ministry of Education, the China Postdoctoral Science Foundation under No. 2024M762829, the Zhejiang Provincial Natural Science Foundation under No. LD24F020002, the "Pioneer" and "Leading Goose" R&D Program of Zhejiang under 2025C01082 and 2025C02263, and the Zhejiang Provincial Priority-Funded Postdoctoral Research Project under No. ZJ2024001.

REFERENCES

[1] Hack3d: The web3 security report 2023. https://www.certik.com/resources/blog/7BokMhPUgffqEvyvXgHNaq-hack3d-the-web3-security-report-2023, 2023.

[2] Code4rena. https://code4rena.com, 2024.

[3] Cryptocurrency prices, market cap and charts - digital assets. https://www.forbes.com/digital-assets/crypto-prices, 2024.

[4] Immunefi. https://immunefi.com, 2024.

[5] Mythril. https://github.com/ConsenSys/mythril, 2024.

[6] SunWeb3Sec/DeFiHackLabs. https://github.com/SunWeb3Sec/DeFiHackLabs, 2024.

[7] Rohan Anil, Sebastian Borgeaud, Yonghui Wu, Jean-Baptiste Alayrac, Jiahui Yu, Radu Soricut, Johan Schalkwyk, Andrew M. Dai, Anja Hauth, Katie Millican, David Silver, Slav Petrov, Melvin Johnson, Ioannis Antonoglou, Julian Schrittwieser, Amelia Glaese, Jilin Chen, Emily Pitler, Timothy P. Lillicrap, Angeliki Lazaridou, Orhan Firat, James Molloy, Michael Isard, Paul Ronald Barham, Tom Hennigan, Benjamin Lee, Fabio Viola, Malcolm Reynolds, Yuanzhong Xu, Ryan Doherty, Eli Collins, Clemens Meyer, Eliza Rutherford, Erica Moreira, Kareem Ayoub, Megha Goel, George Tucker, Enrique Piqueras, Maxim Krikun, Iain Barr, Nikolay Savinov, Ivo Danihelka, Becca Roelofs, Anaïs White, Anders Andreassen, Tamara von Glehn, Lakshman Yagati, Mehran Kazemi, Lucas Gonzalez, Misha Khalman, Jakub Sygnowski, and et al. Gemini: A family of highly capable multimodal models. *CoRR*, abs/2312.11805, 2023.

[8] Priyanka Bose, Dipanjan Das, Yanju Chen, Yu Feng, Christopher Kruegel, and Giovanni Vigna. SAILFISH: vetting smart contract state-inconsistency bugs in seconds. In *43rd IEEE Symposium on Security and Privacy, SP 2022*, pages 161–178. IEEE, 2022.

[9] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. In *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020*, 2020.

[10] Jaeseung Choi, Doyeon Kim, Soomin Kim, Gustavo Grieco, Alex Groce, and Sang Kil Cha. SMARTIAN: enhancing smart contract fuzzing with static and dynamic data-flow analyses. In *36th IEEE/ACM International Conference on Automated Software Engineering, ASE 2021*, pages 227–239. IEEE, 2021.

[11] Yue Duan, Xin Zhao, Yu Pan, Shucheng Li, Minghao Li, Fengyuan Xu, and Mu Zhang. Towards automated safety vetting of smart contracts in decentralized applications. In Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi, editors, *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, CCS 2022, Los Angeles, CA, USA, November 7-11, 2022*, pages 921–935. ACM, 2022.

[12] Thomas Durieux, João F. Ferreira, Rui Abreu, and Pedro Cruz. Empirical review of automated analysis tools on 47, 587 ethereum smart contracts. In Gregg Rothermel and Doo-Hwan Bae, editors, *ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020*, pages 530–541. ACM, 2020.

[13] Josselin Feist, Gustavo Grieco, and Alex Groce. Slither: a static analysis framework for smart contracts. In *Proceedings of the 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain, WETSEB@ICSE 2019*, pages 8–15. IEEE / ACM, 2019.

[14] Neville Grech, Michael Kong, Anton Jurisevic, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis. Madmax: surviving out-of-gas conditions in ethereum smart contracts. *Proc. ACM Program. Lang.*, 2(OOPSLA):116:1–116:27, 2018.

[15] Gustavo Grieco, Will Song, Artur Cygan, Josselin Feist, and Alex Groce. Echidna: effective, usable, and fast fuzzing for smart contracts. In *ISSTA '20: 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 557–560. ACM, 2020.

[16] Jingxuan He, Mislav Balunovic, Nodar Ambroladze, Petar Tsankov, and Martin T. Vechev. Learning to fuzz from symbolic execution with application to smart contracts. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019*, pages 531–548. ACM, 2019.

[17] Bo Jiang, Ye Liu, and W. K. Chan. Contractfuzzer: fuzzing smart contracts for vulnerability detection. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*, pages 259–269. ACM, 2018.

[18] Sukrit Kalra, Seep Goel, Mohan Dhawan, and Subodh Sharma. ZEUS: analyzing safety of smart contracts. In *25th Annual Network and Distributed System Security Symposium, NDSS 2018*. The Internet Society, 2018.

[19] Johannes Krupp and Christian Rossow. teether: Gnawing at ethereum to automatically exploit smart contracts. In *27th USENIX Security Symposium, USENIX Security 2018*, pages 1317–1333. USENIX Association, 2018.

[20] Haonan Li, Yu Hao, Yizhuo Zhai, and Zhiyun Qian. Assisting static analysis with large language models: A chatgpt experiment. In Satish Chandra, Kelly Blincoe, and Paolo Tonella, editors, *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2023*, pages 2107–2111. ACM, 2023.

[21] Ye Liu and Yi Li. Invcon: A dynamic invariant detector for ethereum smart contracts. In *37th IEEE/ACM International Conference on Automated Software Engineering, ASE 2022*, pages 160:1–160:4. ACM, 2022.

[22] Ye Liu, Yue Xue, Daoyuan Wu, Yuqiang Sun, Yi Li, Miaolei Shi, and Yang Liu. Propertygpt: Llm-driven formal verification of smart contracts through retrieval-augmented property generation. In *32nd Annual Network and Distributed System Security Symposium, NDSS 2025, San Diego, California, USA, February 24-28, 2025*. The Internet Society, 2025.

[23] Zhenguang Liu, Peng Qian, Xiang Wang, Lei Zhu, Qinming He, and Shouling Ji. Smart contract vulnerability detection: From pure neural network to interpretable graph feature and expert pattern fusion. In *Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence, IJCAI 2021*, pages 2751–2759. ijcai.org, 2021.

[24] Zhenguang Liu, Peng Qian, Xiaoyang Wang, Yuan Zhuang, Lin Qiu, and Xun Wang. Combining graph neural networks with expert knowledge for smart contract vulnerability detection. *IEEE Trans. Knowl. Data Eng.*, 35(2):1296–1310, 2023.

[25] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. Making smart contracts smarter. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 254–269. ACM, 2016.

[26] Mark Mossberg, Felipe Manzano, Eric Hennenfent, Alex Groce, Gustavo Grieco, Josselin Feist, Trent Brunson, and Artem Dinaburg. Manticore: A user-friendly symbolic execution framework for binaries and smart contracts. In *34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019*, pages 1186–1189. IEEE, 2019.

[27] Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll L. Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, John Schulman, Jacob Hilton, Fraser Kelton, Luke Miller, Maddie Simens, Amanda Askell, Peter Welinder, Paul F. Christiano, Jan Leike, and Ryan Lowe. Training language models to follow instructions with human feedback. In *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022*, 2022.

[28] Anton Permenev, Dimitar K. Dimitrov, Petar Tsankov, Dana Drachsler-Cohen, and Martin T. Vechev. Verx: Safety verification of smart contracts. In *2020 IEEE Symposium on Security and Privacy, SP 2020, 2020*, pages 1661–1677. IEEE, 2020.

[29] Ben Prystawski, Michael Li, and Noah D. Goodman. Why think step by step? reasoning emerges from the locality of experience. In *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023*, 2023.

[30] Chaofan Shou, Shangyin Tan, and Koushik Sen. Ityfuzz: Snapshot-based fuzzer for smart contract. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2023*, pages 322–333. ACM, 2023.

[31] Sunbeom So, Seongjoon Hong, and Hakjoo Oh. Smartest: Effectively hunting vulnerable transaction sequences in smart contracts through language model-guided symbolic execution. In *30th USENIX Security Symposium, USENIX Security 2021*, pages 1361–1378. USENIX Association, 2021.

[32] Sunbeom So, Myungho Lee, Jisu Park, Heejo Lee, and Hakjoo Oh. VERISMART: A highly precise safety verifier for ethereum smart

contracts. In *2020 IEEE Symposium on Security and Privacy, SP 2020*, pages 1678–1694. IEEE, 2020.

[33] Yuqiang Sun, Daoyuan Wu, Yue Xue, Han Liu, Haijun Wang, Zhengzi Xu, Xiaofei Xie, and Yang Liu. Gptscan: Detecting logic vulnerabilities in smart contracts by combining gpt with program analysis. In *46th IEEE/ACM International Conference on Software Engineering, ICSE 2024*, 2024.

[34] Bryan Tan, Benjamin Mariano, Shuvendu K. Lahiri, Isil Dillig, and Yu Feng. Soltype: refinement types for arithmetic overflow in solidity. *Proc. ACM Program. Lang.*, 6(POPL):1–29, 2022.

[35] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurélien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. Llama: Open and efficient foundation language models. In *arXiv preprint arXiv:2302.13971*, 2023.

[36] Petar Tsankov, Andrei Marian Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Bünzli, and Martin T. Vechev. Securify: Practical security analysis of smart contracts. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*, pages 67–82. ACM, 2018.

[37] Sally Junsong Wang, Kexin Pei, and Junfeng Yang. Smartinv: Multimodal learning for smart contract invariant inference. In *2024 IEEE Symposium on Security and Privacy (SP)*, pages 125–125. IEEE Computer Society, may 2024.

[38] Shuai Wang, Chengyu Zhang, and Zhendong Su. Detecting nondeterministic payment bugs in ethereum smart contracts. *Proc. ACM Program. Lang.*, 3(OOPSLA):189:1–189:29, 2019.

[39] Yuepeng Wang, Shuvendu K. Lahiri, Shuo Chen, Rong Pan, Isil Dillig, Cody Born, Immad Naseer, and Kostas Ferles. Formal verification of workflow policies for smart contracts in azure blockchain. In *Verified Software. Theories, Tools, and Experiments - 11th International Conference, VSTTE 2019, New York City, NY, USA, July 13-14, 2019, Revised Selected Papers*, volume 12031 of *Lecture Notes in Computer Science*, pages 87–106. Springer, 2019.

[40] Jason Wei, Yi Tay, Rishi Bommasani, Colin Raffel, Barret Zoph, Sebastian Borgeaud, Dani Yogatama, Maarten Bosma, Denny Zhou, Donald Metzler, Ed H. Chi, Tatsunori Hashimoto, Oriol Vinyals, Percy Liang, Jeff Dean, and William Fedus. Emergent abilities of large language models. *Trans. Mach. Learn. Res.*, 2022, 2022.

[41] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H. Chi, Quoc V. Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models. In *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022*, 2022.

[42] Yuxiang Wei, Chunqiu Steven Xia, and Lingming Zhang. Copiloting the copilots: Fusing large language models with completion engines for automated program repair. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2023*, pages 172–184. ACM, 2023.

[43] Valentin Wüstholz and Maria Christakis. Harvey: a greybox fuzzer for smart contracts. In *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1398–1409. ACM, 2020.

[44] Yinxing Xue, Jiaming Ye, Wei Zhang, Jun Sun, Lei Ma, Haijun Wang, and Jianjun Zhao. xfuzz: Machine learning guided cross-contract fuzzing. *IEEE Trans. Dependable Secur. Comput.*, 21(2):515–529, 2024.

[45] Zhuo Zhang, Brian Zhang, Wen Xu, and Zhiqiang Lin. Demystifying exploitable bugs in smart contracts. In *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023*, pages 615–627. IEEE, 2023.

[46] Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shan Wang, Yufei Xue, Lei Shen, Zihan Wang, Andi Wang, Yang Li, Teng Su, Zhilin Yang, and Jie Tang. Codegeex: A pre-trained model for code generation with multilingual benchmarking on humaneval-x. In *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, KDD 2023*, pages 5673–5684. ACM, 2023.

[47] Yuan Zhuang, Zhenguang Liu, Peng Qian, Qi Liu, Xiang Wang, and Qinming He. Smart contract vulnerability detection using graph neural network. In *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI 2020*, pages 3283–3290. ijcai.org, 2020.