Mining Invariants to Enhance Fuzzing for Detecting Functional Bugs in Smart Contracts

XINGSHUANG LIN, Zhejiang University, China
BINBIN ZHAO*, Zhejiang University, China
QINGE XIE, Zhejiang University, China and Georgia Institute of Technology, USA
NA RUAN, Shanghai Jiao Tong Univeristy, China
JILIANG LI, Xi'an Jiaotong University, China
SHOULING JI, Zhejiang University, China

Smart contracts, which are fundamental to DeFi financial systems, are vulnerable to exploitable bugs that can cause substantial monetary losses. A recent study shows that over 80% of these exploitable bugs, primarily functional bugs, evade detection by existing tools. Automatically identifying functional bugs in smart contracts is challenging 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 PromFuzz++, an automated and scalable system for detecting functional bugs in smart contracts. In PromFuzz++, we first propose a dual-agent framework to identify potentially vulnerable functions for further analysis. We then implement a dual-stage coupling approach that generates invariant checkers using logic information extracted from these functions, enhancing bug-oriented fuzzing on target contracts. Our evaluation shows that PromFuzz++ achieves 86.96% recall and a 93.02% F1-score, improving both metrics by at least 50% over state-of-the-art methods. An in-depth analysis of 10 real-world DeFi projects uncovered 30 zero-day bugs, 24 of which have been assigned CVE IDs.

CCS Concepts: • **Software and its engineering → Software verification and validation**; • **Security and privacy → Software security engineering**.

Additional Key Words and Phrases: Smart Contract, Functional Bug Detection, Blockchain, Large Language Model

1 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].

*Binbin Zhao is the corresponding author.

Authors' Contact Information: Xingshuang Lin, cs.xslin@zju.edu.cn, Zhejiang University, Hangzhou, China; Binbin Zhao, binbinz@zju.edu.cn, Zhejiang University, Hangzhou, China; Qinge Xie, xqgtiti@gmail.com, Zhejiang University, Hangzhou, China and Georgia Institute of Technology, Atlanta, USA; Na Ruan, naruan@cs.sjtu.edu.cn, Shanghai Jiao Tong Univeristy, Shanghai, China; Jiliang Li, jiliang.li@xjtu.edu.cn, Xi'an Jiaotong University, Xi'an, China; Shouling Ji, sji@zju.edu.cn, Zhejiang University, Hangzhou, China.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2026 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM 1557-7392/2026/9-ART
https://doi.org/10.1145/3849091

Just Accepted

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 [17, 42], fuzzing [14, 34], verification [36, 39], and symbolic execution [5, 35]. Despite these efforts, a recent study [50] 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.

1.1 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.

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.

1.2 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,

Just Accepted

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.

1.3 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 564 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 5 dimensions using 7 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.

2 Background

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

2.1 Smart Contract

Smart contracts are blockchain-deployed programs that maintain persistent on-chain states and execute their encoded logic when invoked by transactions or calls from other contracts [7]. They provide a programmable mechanism for implementing application logic on blockchains without relying on a centralized server. Smart contracts have been used in a wide range of decentralized applications, such as decentralized finance and governance. They can be implemented in different programming languages, such as Solidity [8] and Vyper [9].

This paper focuses on smart contracts written in Solidity. Solidity is an object-oriented, high-level programming language for implementing smart contracts and is designed to target the Ethereum Virtual Machine (EVM) [8]. Accordingly, our target contracts are Solidity smart contracts deployed on Ethereum or other EVM-compatible

Just Accepted

blockchains. A Solidity contract typically consists of state variables, functions, modifiers, and events. State variables record persistent contract data, functions define executable behaviors, modifiers specify reusable access or validity checks, and events are used to log execution information. Unless otherwise specified, we use the term "smart contract" to refer specifically to Solidity smart contracts in this paper.

2.2 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 [38, 50].

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.

2.3 Large Language Models

Large Language Models (LLMs) have demonstrated significant advancements across a variety of tasks, including code generation [51], static analysis [24], and program repair [47]. Currently, a set of prominent LLMs have been developed, such as the Generative Pre-trained Transformer (GPT) by OpenAI, LLaMA [40] by Meta, and Gemini [10] 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 [45] that set them apart from traditional pre-trained models. They have three notable emergent abilities: in-context learning [12], instruction following [31], and step-by-step reasoning [33]. 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) [46] 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

Just Accepted

contracts. We focus on designing novel prompt strategies to trigger its emergent capabilities and implement practical measures to enhance its output robustness.

2.4 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 = 10 * 1e18, a user providing 50 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 amountInShares (equal to 5), rather than the correct variable amount (equal to 50). 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* [38] and *SMARTINV* [43], 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,

Just Accepted

_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.

3 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 LLM 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.

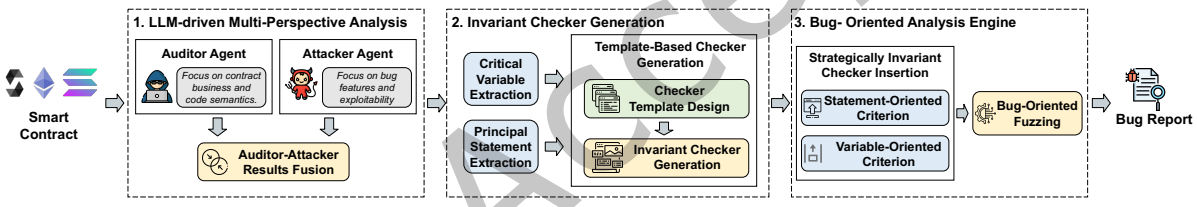


Fig. 2. Framework of PromFuzz++.

3.1 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 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 strategy. This framework incorporates two specialized agents: the Auditor Agent and the Attacker Agent, each designed to enhance the emergent capabilities of LLM 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

Just Accepted

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 framework enables precise identification and reliable assessment of potential bugs, thereby enhancing the robustness of LLM's output.

In our dual-agent prompt strategy, 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 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* [38]. 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. [50]. 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, LLM is informed that it is participating in a code auditing exercise. We then allow LLM 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 LLM confirms a potential issue, we follow up with specific questions about the vulnerable code attributes

Just Accepted

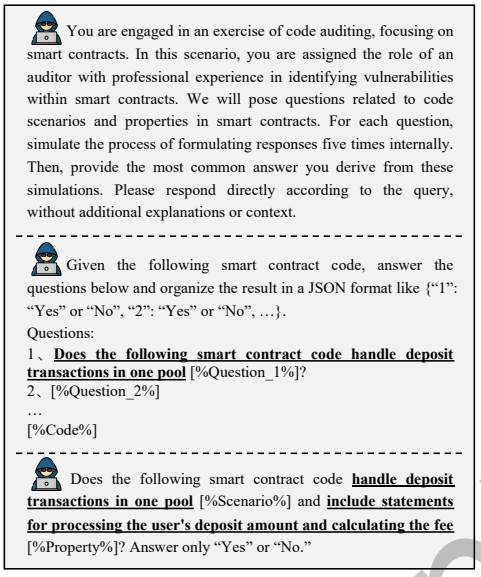


Fig. 3. Prompt template for the Auditor Agent.

or operations—i.e., the property. If LLM 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 LLM 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 Auditor Agent, LLM is informed that it is participating in an ethical hacking exercise. Next, the target smart contract code is inputted, and LLM 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 LLM. We then query LLM whether the identified features could lead to exploitable attacks based on the given attack model. An affirmative response from LLM indicates the presence of a functional bug, while a negative response helps us dismiss that particular bug.

Just Accepted

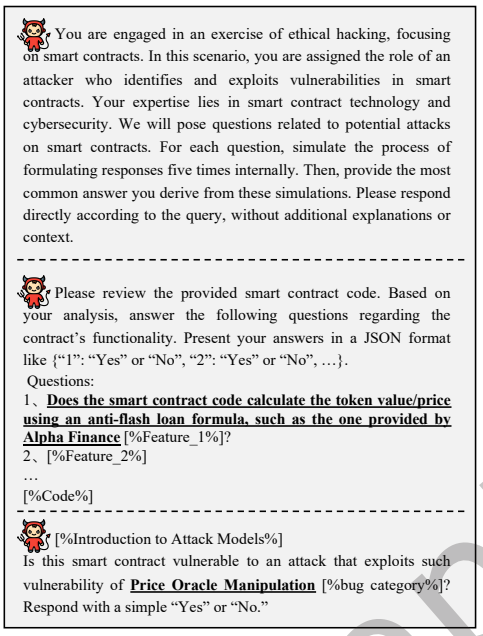


Fig. 4. Prompt template for the Attacker Agent.

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 rules for results fusion are delineated as follows:

As shown in Algorithm 1, consider a scenario where for each analyzed code segment, the Attacker Agent identifies the primary bug categories $P = \{p_j\}$ at line 1, and the Auditor Agent identifies the subcategories $S = \{s_i\}$ at line 2. The fusion process iterates over each subcategory $s_i \in S$ and each primary category $p_j \in P$ to examine all Auditor–Attacker result pairs. The fusion process operates under the following rules:

1) If s_i is a subcategory of p_j, confirm s_i as a candidate type and add the merged result to B (line 6-8).
2) In addition to confirming s_i, if s_i belongs to p_j, further include all other subcategories of p_j as candidate types to address potential overlaps or ambiguities (line 9-11).
3) If s_i does not correspond to any subcategory of p_j, then exclude s_i from the candidate types (line 14).
4) If either S or P is empty, no valid Auditor–Attacker result pair can be formed, and the current segment is skipped without generating candidate types (lines 4–5).

For primary categories characterized by distinctly different subcategory features, such as *insecure calculating logic* and *incorrect control mechanism*, we apply rules 1), 3), and 4). For the other primary categories, we implement rules 2), 3), and 4). This selective application of fusion rules helps minimize false positives while maximizing the conversion of potential false negatives into true positives by the Auditor Agent. Although this method may initially increase the number of false positives, subsequent bug-oriented fuzzing processes are designed to effectively eliminate these inaccuracies.

Just Accepted

Algorithm 1: Auditor-Attacker Results Fusion

Input: C: smart contract code
Output: B: potential functional bug subcategory candidates

$P \leftarrow Attacker_Agent(C)$ ▷ Primary-category result list
$S \leftarrow Auditor_Agent(C)$ ▷ Subcategory result list
$B \leftarrow [\,]$ ▷ Initialize the result list
for $s_i \in S$ **do**
  for $p_j \in P$ **do** ▷ Examine all Auditor–Attacker result pairs
    if s_i is a subcategory of p_j **then**
      $r_{mer} \leftarrow Merge(p_j, s_i)$
      $B \leftarrow B.append(r_{mer})$
      if $j \in \{1, 2\}$ **then**
        $r_{mut} \leftarrow Mutation(p_j, s_i)$
        $B \leftarrow B.append(r_{mut})$
      end if
    else
      Continue
    end if
  end for
end for
return B

Figure 5 illustrates an example of how our fusion rules are applied. First, the Auditor Agent identifies the code scenario as a token transfer and flags the property of this transfer as unsafe, classifying it under the subcategory of unauthorized transfer. Subsequently, the Attacker Agent analyzes potential bug features, indicating that these bugs could be exploited and categorizing them under the primary category of unauthorized behavior, which includes two subcategories: unauthorized transfer and approval not clear. Finally, we apply rule 2), 3) and 4) to integrate their analyses, leading to the identification of unauthorized transfer and approval not clear. The true positive identified in this code is approval not clear. Without this integration, we would fail to recognize this bug.

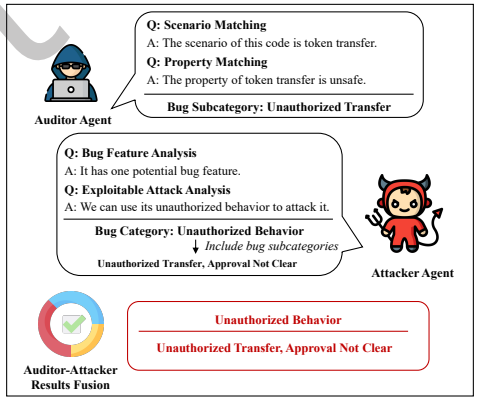


Fig. 5. Auditor-Attacker results fusion process corresponds to the motivating example.

Just Accepted

3.2 Invariant Checker Generation

While we have identified potentially vulnerable functions, we cannot solely rely on LLM'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 LLM'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.

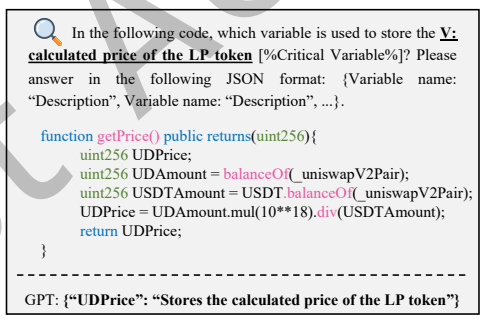


Fig. 6. Prompt template for extracting critical variables from code snippets potentially vulnerable to AMM price oracle manipulation.

In response to this challenge, we propose a hierarchical matching method that capitalizes on the capabilities of LLM 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 Table 1. Second, leveraging the identified

Just Accepted

elements, we formulate specific prompts and input them into LLM to query each vulnerable function as identified by the LLM-driven multi-perspective analysis module. Figure 6 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, LLM is able to precisely pinpoint the relevant critical variables and principal statements, enhancing the accuracy of our bug detection process.

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

Fig. 7. The checker for AMM price oracle manipulation.

Table 1. Characteristics of critical variables and principal statements for each bug subcategory, along with the associated invariant checkers.

Bug Subcategory	Key Variables (V) & Statements (S)	Invariant Checker
AMM Price Oracle Manipulation	V1: calculated price of the LP token	PriceChange_Checker(V1)
Non-AMM Price Oracle Manipulation	V1: the amount of tokens spent by the user before the swap V2: the amount of tokens received by the user after the swap	ExchangeRate_Checker(V1,V2)
Approval Not Clear	V1: sending address for token transfers V2: authorization array for token transfers V3: the amount of tokens being transferred	TokenChange_Checker(V1,V2,V3)
Unauthorized Transfer	V1: sending address for token transfers V2: authorization array for token transfers V3: the amount of tokens being transferred	TokenChange_Checker(V1,V2,V3)
Wrong Checkpoint Order	S1: the first to record the checkpoint S2: the first to perform a calculation or update on the tokens	StatementOrder_Checker(S1,S2)
Risky First Deposit	V1: the total supply of tokens V2: the share that a user can obtain V3: the amount of tokens spent by the user	ShareSafety_Checker(V1,V2,V3)
Wrong Interest Rate Order	S1: the first to update the interest rate S2: the first to use the interest rate to calculate or update the token balance, rewards, or shares	StatementOrder_Checker(S1,S2)
Improper Handling of Deposit Fee	V1: the amount of tokens spent by the user without any deductions V2: the net amount added to the user's balance after deducting the deposit fee V3: the amount of deposit fee	TokenChange_Checker(V1,V2,V3)
Wrong Implementation of Amount Lock	V1: visitng the user's original locked amount V2: the amount by which the user's locked funds have increased or decreased V3: stores the new locked amount after increasing or decreasing based on the isIncrease flag	TokenChange_Checker(V1,V2,V3)
Vote Manipulation	V1: contract variable for voting tokens V2: the amount of voting tokens cast by the user in this round S1: the first to execute a voting action	StateChange_Checker(V1,V2,S1)

Just Accepted

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, for each type of bug, we design a corresponding invariant checker template that targets the vulnerable features of different bugs, as detailed in Table 1. We mainly design 6 types of checkers: **PriceChange_Checker**, **ExchangeRate_Checker**, **TokenChange_Checker**, **StatementOrder_Checker**, **ShareSafety_Checker**, and **StateChange_Checker**. Each checker contains multiple invariant snippets that encapsulate the conditional statements for assessing contract state and must be inserted at specific locations within the smart contract. For instance, as shown in Figure 7, 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.

3.3 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.

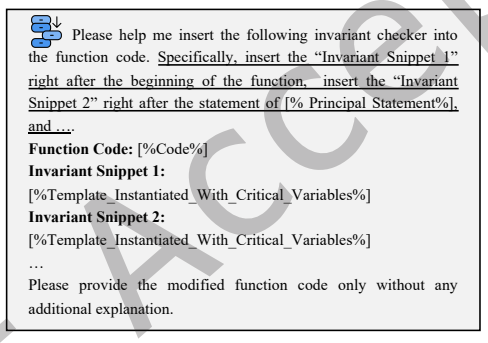

Please help me insert the following invariant checker into the function code. Specifically, insert the "Invariant Snippet 1" right after the beginning of the function, insert the "Invariant Snippet 2" right after the statement of [% Principal Statement%], and ….
Function Code: [%Code%]
Invariant Snippet 1:
[%Template_Instantiated_With_Critical_Variables%]
Invariant Snippet 2:
[%Template_Instantiated_With_Critical_Variables%]
…
Please provide the modified function code only without any additional explanation.

Fig. 8. Prompt template for invariant checker insertion.

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. We leverage an LLM to operationalize the above two criteria. Figure 8 illustrates the prompt template designed to guide the insertion of the invariant checker. The prompt provides explicit instructions, such as "after the beginning of the function" and "before or after the principal statement", so that LLM can accurately align each invariant snippet with its designated insertion point. Consequently, LLM produces a modified function with the invariant checker correctly integrated.

Just Accepted

Algorithm 2: 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** $t \neq \emptyset$ **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.

3.4 Functional Bug Detection

The functional bug detection module aims to perform comprehensive bug detection on smart contracts. Algorithm 2 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. This report serves as a crucial tool for developers to understand and rectify bugs in their smart contracts.

4 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 strategy contribute to the performance of PromFuzz++? (Section 4.3)
RQ2: What is PromFuzz++'s accuracy in generating and inserting invariant checkers? (Section 4.4)

Just Accepted

RQ3: How effective is PromFuzz++ at detecting functional bugs in smart contracts? (Section 4.5)
RQ4: How robust is PromFuzz++ under LLM-induced randomness and error propagation across its pipeline? (Section 4.6)
RQ5: Is PromFuzz++ capable of discovering zero-day functional bugs in real-world smart contracts? (Section 4.7)

4.1 Experimental Setup

Environment and Configuration. We use GPT-4-Turbo through the OpenAI API with the default parameter settings for all LLM-assisted tasks. The fuzzing time budget is set to one hour for each case. All experiments were conducted on a server equipped with two Intel Xeon Platinum 8380 CPUs (80 physical cores in total) and 256 GB of RAM, running Ubuntu 20.04.6 LTS.
Evaluation Metrics. For the vulnerability detection experiments, TP, FP, and FN are counted for each reported bug category within each case. For each case, a reported bug category matching the labeled ground-truth category is counted as a TP, while each non-matching reported category is counted as an FP. Failure to report the ground-truth category is counted as an FN. Since a detector may report multiple bug categories for a single case, one case can contribute multiple FPs.

4.2 Dataset

To ensure a comprehensive assessment of PromFuzz++'s effectiveness, we systematically collect 767 smart contracts from various sources, including DeFiHackLabs [6], Web3bugs [50], SmartBugs [16], GPTScan [38], VeriSmart [36], on-chain smart contracts, and real-world DeFi projects. After further analysis, we remove 203 contracts that either lack functional bugs or are missing essential external files or configuration information required for compilation. Ultimately, we construct 7 datasets based on the remaining 564 contracts, covering a broad spectrum of functional bug categories and evaluation perspectives.

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.

Insertion-challenging smart contract dataset (D_{IC-SC}) consists of smart contracts with deliberately complex program structures around invariant checker insertion locations. The purpose of this dataset is to evaluate whether PromFuzz++ can more accurately insert invariant checkers than PromFuzz under challenging control-flow and data-dependency patterns. D_{IC-SC} includes 10 cases derived from $D_{Exploit-SC}$, D_{BV-SC}, and $D_{Synth-SC}$, involving 51 contracts in total. These cases are randomly selected from the three datasets and manually transformed by introducing additional conditional branches, early-return paths, and variable-scope changes, while preserving the original vulnerable behavior and expected triggering conditions.

Just Accepted

Bug-free smart contract dataset (D_{BF-SC}) includes well-audited and widely used smart contracts that are free from any known functional bugs. These contracts serve as a benchmark to assess the false positive rate of PromFuzz++. Specifically, D_{BF-SC} consists of 303 open-source smart contracts from 6 mainstream Ethereum-compatible chains.

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 84 extracted critical variables and principal statements across 36 smart contracts. 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 2. Evaluation datasets and contract-level SLOC statistics.

Dataset	# Evaluation Targets	# Contracts	SLOC Statistics					RQ
			Total	Mean	Median	Min	Max	
$D_{Exploit-SC}$	7 cases	47	6,816	145	45	4	1,941	RQ1,RQ3,RQ4
D_{BV-SC}	10 cases	36	10,517	292	118	4	1,056	RQ1,RQ3,RQ4
$D_{Synth-SC}$	6 cases	44	1,534	35	21	4	164	RQ1,RQ3,RQ4
D_{IC-SC}	10 cases	51	8,023	157	45	4	923	RQ3,RQ4
D_{BF-SC}	303 contracts	303	88,568	292	231	27	4,443	RQ1
D_{SE-SC}	36 functions	36	646	18	15	8	50	RQ2
$D_{Real-DeFi}$	10 cases	108	16,490	153	86	6	907	RQ5

Summary: Overall, $D_{Exploit-SC}$, D_{BV-SC}, and $D_{Synth-SC}$ constitute the complete set of labeled cases used in our main functional bug detection evaluation. Over these three datasets, Price Oracle Manipulation accounts for 21.7% of the cases, Unauthorized Behavior accounts for 43.5%, Insecure Calculating Logic accounts for 21.7%, and Incorrect Control Mechanism accounts for 13.0%. This distribution shows that our evaluation covers multiple representative functional bug categories rather than focusing on a single dominant type. Although D_{IC-SC} also contains vulnerable cases, it is derived from the above datasets and is used specifically to evaluate invariant-checker insertion under challenging program structure. Therefore, we report it separately to avoid double counting. The remaining datasets are designed for targeted evaluations from different perspectives, including dual-agent analysis, symbol extraction accuracy, and real-world DeFi project assessment. The contract-level SLOC statistics, including the total, mean, median, minimum, and maximum values, are reported in Table 2.

4.3 Dual-Agent Architecture Evaluation

This subsection addresses the effectiveness of the dual-agent prompt strategy of PromFuzz++. First, we compare the Dual-Agent version of PromFuzz++ with an Auditor-Agent-only variant, as both variants produce subcategory-level bug outputs compatible with the downstream stages. We also report the standalone performance of the Attacker-Agent, which produces primary-category bug outputs and thus cannot be directly integrated into the subcategory-level pipeline. This evaluation is conducted on three datasets: $D_{Exploit-SC}$, D_{BV-SC}, and $D_{Synth-SC}$. Second, we compare Dual-Agent architecture with representative reasoning models to examine

Just Accepted

whether our workflow can outperform the automatically planned reasoning processes of such models. We select two mainstream reasoning models: DeepSeek-R1 and OpenAI-o3, and evaluate them on the same three datasets. Third, we evaluate the false positive rate of Dual-Agent architecture on D_{BF-SC}. Although PromFuzz++ reduces false positives during the fuzzing phase, any additional false positives still impose overhead, as PromFuzz++ must validate bugs that do not exist. We further compare Dual-Agent architecture with existing LLM-based reasoning architectures using the false positive rate as the metric. We construct two baseline architectures, Gr-Agent and Sr-Agent, extracted respectively from *GPTScan* [38] and *SMARTINV* [43].

Ablation Study. As shown in Table 3, Dual-Agent architecture achieves substantial performance improvements over the Auditor-Agent architecture. Both architectures produce subcategory-level bug outputs that are compatible with the downstream stages of PromFuzz++. Specifically, Dual-Agent architecture identifies 21 true positives with only 2 false negatives, achieving a recall of 91.30% and an F1-score of 53.85% across $D_{Exploit-SC}$, D_{BV-SC}, and $D_{Synth-SC}$. In contrast, Auditor-Agent architecture identifies 15 true positives and 8 false negatives, achieving a recall of 65.22% and an F1-score of 48.39%. These results show that incorporating the attacker perspective improves the coverage of subcategory-level functional bug detection. We also report the standalone performance of Attacker-Agent. Since Attacker-Agent produces only primary-category bug outputs, its results cannot be directly integrated into the subcategory-level downstream pipeline and should not be regarded as an end-to-end ablation variant. Nevertheless, its standalone performance shows that it can effectively identify potential attack directions, achieving a precision of 59.46%, a recall of 95.65%, and an F1-score of 73.33% at the primary bug category level.

Table 3. Comparison of Dual-Agent, Auditor-Agent, and Attacker-Agent.

Architecture	Bug Output Granularity	Downstream Compatibility	Overall				
			TP	FP	FN	Recall	F1-Score
Dual-Agent	Subcategory	YES	**21**	**34**	**2**	**91.30%**	**53.85%**
Auditor-Agent	Subcategory	YES	15	24	8	65.22%	48.39%
Attacker-Agent	Primary category	NO	22	15	1	95.65%	73.33%

Note: Downstream Compatibility indicates whether the bug output granularity of the architecture matches the input granularity required by the downstream stages of PromFuzz++.

We further analyze the improvement of Dual-Agent architecture and the reasons for false positives and false negatives. The improvement mainly comes from the complementary roles of the two agents. Auditor-Agent performs fine-grained inspections of contract semantics, business scenarios, and logical correctness, which enables it to produce subcategory-level bug outputs required by the downstream stages. However, relying only on this auditing perspective may lead to underreporting, especially for bugs whose risks become evident only under specific exploit scenarios. In contrast, Attacker-Agent focuses on coarse-grained exploitability analysis and examines whether contract behaviors can be exploited to achieve unintended outcomes. This attack-oriented perspective helps Dual-Agent architecture discover more potential functional bugs and reduce false negatives. However, the broader exploration of Attacker-Agent may also over-generate candidates, leading to additional false positives. These false positives are less harmful in PromFuzz++ because they can be further filtered by the subsequent validation stages. In addition, LLM misinterpretation can also introduce false positives and false negatives, further affecting the performance of architectures.

Comparison with Reasoning Models. As shown in Table 4, Dual-Agent architecture outperforms mainstream reasoning model architectures. Specifically, Dual-Agent architecture ahieves a recall of 91.30% and an F1-score of

Just Accepted

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 4. 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%

False Positive Rate Evaluation. Both Dual-Agent architecture and Gr-Agent architecture can analyze whether a function contains functional bugs, operating at the function-level, while Sr-Agent architecture is limited to file-level analysis. For a fair comparison, we conduct evaluations at both the function and file levels. As illustrated in Table 5, Dual-Agent architecture has a slight advantage with false positive rates of 6.31% and 15.38%, respectively, than Gr-Agent architecture. Despite strictly adhering to the guidelines for Sr-Agent architecture and utilizing its publicly released code, Sr-Agent architecture records a significantly higher false positive rate of 72% at file-level.

Table 5. False positive rate evaluation.

Architecture	D_{BF-SC}	
	FPR (Function-Level)	**FPR (File-Level)**
Dual-Agent	**6.31%**	**15.38%**
Gr-Agent	7.16%	15.75%
Sr-Agent	N/A	72%

We further analyze the false positives of these tools. For Sr-Agent architecture, the primary focus of its prompt design is on generating critical invariants. However, this approach lacks the comprehensive reasoning capabilities necessary for effective potential bug detection. This shortcoming in the analytical strategy significantly contributes to the tool's notably high false positive rate. For Gr-Agent architecture, the primary reason for false positives arises from its single-agent design, which can lead to the incorrect identification of correct behaviors as functional bugs, such as those involving insecure calculating logic. Although Dual-Agent architecture has similar reasons, its innovative Dual-Agent design effectively mitigates a greater number of false positives compared to Gr-Agent architecture.

RQ1: Dual-Agent architecture exhibits strong effectiveness in pinpointing potential functional bugs, outperforming both the ablated variant and the reasoning models. Moreover, Dual-Agent architecture significantly mitigates false positives, improving the overall efficiency of PromFuzz++.

Just Accepted

4.4 Invariant Checker Evaluation

This subsection answers RQ2. First, we evaluate the accuracy of PromFuzz++ in extracting critical variables and principal statements from potentially vulnerable functions, which provide the basis for generating and inserting invariant snippets. Next, using the extracted variables and statements, we measure the success rate of invariant snippet generation and insertion, which reflects whether the generated invariant checker can be correctly integrated into the target contract. Moreover, since PromFuzz++ improves the invariant checker insertion strategy over PromFuzz, we conduct a direct comparison between PromFuzz++ and PromFuzz to quantify the effectiveness of the improved insertion strategy. We conduct all the above evaluations on D_{SE-SC}.

Variables and Statements Extraction. As shown in Table 6, PromFuzz++ demonstrates excellent performance in both critical variable and principal statement extraction, achieving an overall precision of 96.43%. Specifically, for the variable extraction task, PromFuzz++ achieves a precision of 97.40%, with two false positives. The false positives are due to unconventional variable naming. For the statement extraction task, it achieves a precision of 85.71%, 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 complex function.

Table 6. Variables and statements extraction accuracy.

Task	D_{SE-SC}		
	TP	**FP**	**Precision**
Variables Extraction	75	2	97.40%
Statements Extraction	6	1	85.71%
Overall	**81**	**3**	**96.43%**

Invariant Checker Generation and Integration. These critical variables and principal statements are expected to yield 64 invariant snippets corresponding to 36 invariant checkers. As shown in Table 7, PromFuzz++ successfully generates 63 invariant snippets with 1 failure, achieving a success ratio of 98.44%. Using these generated snippets, PromFuzz++ then performs snippet insertion and achieves 62 successful insertions with 1 failure, achieving a success ratio of 98.41%. In total, PromFuzz++ successfully integrates 34 invariant checkers into the target contracts, achieving an overall success ratio of 94.44%. We compare PromFuzz++ with PromFuzz to evaluate the effectiveness of the LLM-based insertion strategy. Since PromFuzz++ and PromFuzz use the same invariant snippet generation module, they achieve the same result in invariant snippet generation, with 63 successful generations and 1 failure. The main difference lies in the subsequent insertion process. PromFuzz++ successfully inserts 62 invariant snippets with a success ratio of 98.41%, whereas PromFuzz inserts 52 invariant snippets with a success ratio of 82.54%. This improvement further increases the invariant checker integration success ratio by 27.77 percentage points compared with PromFuzz.

Table 7. Invariant checker task evaluation accuracy.

Task	PromFuzz++			PromFuzz		
	# Success	**# Failure**	**Success Ratio**	**# Success**	**# Failure**	**Success Ratio**
Invariant Snippet Generation	63	1	98.44%	63	1	98.44%
Invariant Snippet Insertion	62	1	98.41%	52	11	82.54%
Invariant Checker Integration	**34**	**2**	**94.44%**	**24**	**12**	**66.67%**

We further analyze the failures in the integration process. For PromFuzz++, the one failure in invariant snippet generation arises from the incorrect extraction of critical variables, which prevents valid snippet generation.

Just Accepted

The one failure in invariant snippet insertion is caused by a failure in extracting the corresponding principal statement, making it impossible to determine the required insertion point. In contrast, the failures of PromFuzz mainly come from its rule-based insertion method. When facing complex program structures, such as early-return paths, and variable-scope changes, the rule-based method may fail to identify whether the inserted invariant can be reached and whether the required variables are still in scope. As a result, the inserted invariant may either fail to take effect or introduce compilation errors. These results show that the improved insertion strategy of PromFuzz++ enhances the robustness of invariant checker insertion.

RQ2: PromFuzz++ efficiently extracts highly stable variables and statements, achieving an overall precision of 96.43%. Consequently, PromFuzz++ proves its capability to generate and insert invariant checkers, achieving an overall success ratio of 94.44%. Compared with PromFuzz, PromFuzz++ improves the success ratio by 27.77 percentage points, demonstrating the effectiveness of its LLM-based insertion strategy.

4.5 Functional Bug Detection Accuracy

This subsection answers RQ3. First, we evaluate the end-to-end functional bug detection accuracy of PromFuzz++ on the original functional-bug datasets. Second, we further evaluate PromFuzz++ by comparing it with PromFuzz and PromFuzz-PE. The comparison between PromFuzz++ and PromFuzz assesses whether the LLM-assisted invariant-checker insertion strategy improves insertion effectiveness. The comparison between PromFuzz++ and PromFuzz-PE assesses the effectiveness of the overall strategy built upon the fuzzing engine.

Comparison with Existing Baselines. We evaluate the functional bug detection accuracy of PromFuzz++ using two metrics: recall and F1-score. We compare PromFuzz++ with three off-the-shelf tools: *GPTScan*, *SMARTINV*, and *SMARTIAN* [14]. We do not compare PromFuzz++ with several program verification-based techniques [15, 22, 32], because these approaches primarily target implementation-level properties and usually require manually specified verification constraints, making them less comparable for functional bug detection. We perform them on $D_{Exploit-SC}$, D_{BV-SC}, and $D_{Synth-SC}$. As shown in Table 8, PromFuzz++ demonstrates superior performance in detecting functional bugs across $D_{Exploit-SC}$, D_{BV-SC}, and $D_{Synth-SC}$. Overall, PromFuzz++ detects 20 true positives with 0 false positives and only 3 false negatives, achieving a recall of 86.96% and an F1-score of 93.02%. Further analysis of the false negatives reveals two primary causes. First, one false negative is caused by inaccurate extraction of critical variables and principal statements. Second, the remaining two false negatives occur during the LLM-driven analysis step, where the LLM fails to identify two critical bugs.

Table 8. Functional bug detection accuracy.

Tool	$D_{Exploit-SC}$, D_{BV-SC}, $D_{Synth-SC}$				
	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%
SMARTIAN	3	0	16	15.79%	27.27%

In comparison, *GPTScan* and *SMARTINV* yield similar numbers of true positives and false negatives. However, *SMARTINV* records the highest number of false positives among the evaluated tools assessed. *SMARTIAN*, a fuzzing tool, shows inconsistent performance as it fails to operate on several smart contract files. Consequently,

Just Accepted

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. Furthermore, *SMARTIAN* is limited by its built-in oracle, which lack the capability to identify functional bugs effectively.

To further investigate the performance differences between PromFuzz++ and the LLM-based baselines, we report the 95% Wilson confidence intervals for recall and conduct two-sided exact McNemar tests on $D_{Exploit-SC}$, D_{BV-SC}, and $D_{Synth-SC}$. As shown in Table 9, PromFuzz++ achieves a recall of 86.96% with a 95% Wilson confidence interval of 67.87%–95.46%, substantially higher than the recalls of *GPTScan* and *SMARTINV*, which are 30.43% and 26.09%, with confidence intervals of 15.60%–50.87% and 12.55%–46.47%, respectively. Moreover, the paired exact McNemar tests indicate that the differences between PromFuzz++ and each of the two LLM-based baselines are statistically significant, with $p = 0.00098$ for *GPTScan* and $p = 0.00012$ for *SMARTINV*. These results provide further statistical evidence that PromFuzz++ outperforms the two LLM-based baselines in functional bug detection.

Table 9. Statistical comparison of PromFuzz++ with LLM-based baselines.

Tool	Recall (95% Wilson CI)	Exact McNemar p-value
PromFuzz++	86.96% (67.87%–95.46%)	–
GPTScan	30.43% (15.60%–50.87%)	0.00098
SMARTINV	26.09% (12.55%–46.47%)	0.00012

Note: The p-values are obtained using two-sided exact McNemar tests comparing PromFuzz++ with the corresponding baseline on $D_{Exploit-SC}$, D_{BV-SC}, and $D_{Synth-SC}$.

Comparison with PromFuzz and the Pure Fuzzing Engine. To further evaluate the improvement of PromFuzz++, we conduct an additional comparison involving three variants: PromFuzz++, PromFuzz, and PromFuzz-PE. PromFuzz++ is an enhanced version of PromFuzz. Specifically, PromFuzz++ adopts an LLM-based invariant-checker insertion strategy, whereas PromFuzz relies on a rule-based insertion strategy. PromFuzz-PE denotes the pure-engine variant of PromFuzz++, where only the underlying fuzzing engine is retained. Since PromFuzz++ is built upon *ItyFuzz*, PromFuzz-PE is equivalent to running *ItyFuzz* in our evaluation. In addition to $D_{Exploit-SC}$, D_{BV-SC}, and $D_{Synth-SC}$, we also introduce D_{IC-SC} to evaluate whether PromFuzz++ improves invariant checker insertion over PromFuzz under more challenging program structures. As described in Section 4.2, D_{IC-SC} is derived from the above three datasets by manually introducing more complex program structures around invariant-checker insertion locations while preserving the original vulnerable behaviors and expected triggering conditions.

As shown in Table 10, PromFuzz++ outperforms PromFuzz across the four datasets. Specifically, PromFuzz++ detects 30 true positives with only 3 false negatives, achieving a recall of 90.91% and an F1-score of 95.24%. In contrast, PromFuzz detects 20 true positives with 13 false negatives, achieving a recall of 60.61% and an F1-score of 75.47%. Compared with PromFuzz, PromFuzz++ improves recall by 30.30 percentage points and F1-score by 19.77 percentage points, while reducing false negatives from 13 to 3. Our further analysis shows that this improvement mainly comes from the advantage of LLM-based checker insertion over rule-based insertion. Although both PromFuzz++ and PromFuzz rely on generated invariant checkers to expose functional bugs, PromFuzz inserts checkers according to predefined rules, which are effective only when the insertion context follows regular patterns. When the target location contains more challenging structures, these rules may select

Just Accepted

an incorrect insertion point or place the checker outside the valid scope of key variables. Such insertion errors can make the inserted invariant ineffective, resulting in the failure of functional bug detection. In contrast, PromFuzz++ leverages the LLM-based insertion strategy to reason about the surrounding program context and identify more appropriate insertion locations under complex program structures, thereby improving its detection accuracy.

Table 10. Comparison of PromFuzz++, PromFuzz, and PromFuzz-PE.

Tool	$D_{Exploit-SC}$, D_{BV-SC}, $D_{Synth-SC}$, D_{IC-SC}				
	TP	**FP**	**FN**	**Recall**	**F1-Score**
PromFuzz++	30	0	3	90.91%	95.24%
PromFuzz	20	0	13	60.61%	75.47%
PromFuzz-PE (*ItyFuzz*)	0	0	33	0	0

The comparison between PromFuzz++ and PromFuzz-PE (*ItyFuzz*) further shows that the effectiveness of PromFuzz++ does not merely come from the underlying fuzzing engine. As shown in Table 10, PromFuzz-PE detects 0 true positives, indicating that the underlying fuzzing engine alone is insufficient for detecting functional bugs. Our further analysis shows that this limitation mainly comes from the absence of functional bug oracles. Although PromFuzz-PE can still explore contract states and transaction sequences, its built-in bug oracles are mainly designed for generic security violations rather than functional bugs. Therefore, even if the fuzzing engine reaches a vulnerable state, it lacks the oracles required to recognize the behavior as a functional bug. In contrast, PromFuzz++ generates invariant checkers as functional bug oracles and robustly inserts them into the target contracts. These results show that the improvement of PromFuzz++ comes from the combination of functional invariant construction and LLM-based checker insertion, rather than from the fuzzing engine alone.

RQ3: PromFuzz++ achieves great performance compared to existing tools in detecting functional bugs in smart contracts. Further comparison with PromFuzz and PromFuzz-PE shows that functional invariant checkers and the LLM-based checker insertion strategy play a vital role in effective functional bug detection.

4.6 Robustness Analysis of PromFuzz++

This subsection answers RQ4 by evaluating the robustness of PromFuzz++ from two complementary perspectives. First, to assess the robustness of PromFuzz++ under the randomness introduced by multiple LLM-based reasoning stages, we run PromFuzz++ five times on $D_{Exploit-SC}$, D_{BV-SC}, $D_{Synth-SC}$, and D_{IC-SC}, and report the mean, standard deviation, minimum, and maximum values of recall and F1-score across these runs. Next, we further conduct an LLM-induced error propagation analysis of PromFuzz++ to reveal whether failures are isolated within individual modules or cascade downstream to affect the final results, thereby providing a deeper understanding of the robustness and limitations of PromFuzz++.

Repeated-Run Robustness. As shown in Table 11, PromFuzz++ achieves stable performance across repeated runs. The mean recall is 89.70%, with a standard deviation of 1.66%. The recall varies only within a narrow range, from 87.88% to 90.91%. The mean F1-score is 94.56%, with a standard deviation of 0.93%, ranging from 93.54% to 95.24%. These repeated-run results demonstrate that PromFuzz++ maintains consistent functional bug detection performance despite the inherent randomness introduced by multiple LLM-based reasoning stages, providing evidence for the robustness of PromFuzz++.

Just Accepted

Table 11. Robustness of PromFuzz++ in functional bug detection across five repeated runs

Metric	Mean	Std. Dev.	Min	Max
Recall	89.70%	1.66%	87.88%	90.91%
F1-score	94.56%	0.93%	93.54%	95.24%

Error Propagation Analysis. We further analyze how LLM-induced errors may propagate across PromFuzz++ by examining possible failure cases in each LLM-related stage and assessing their impacts on the final results.

First, the LLM-driven multi-perspective analysis stage serves as the entry point of the functional bug detection pipeline. Its output determines whether a candidate case can enter the downstream analysis. Therefore, false negatives at this stage are more critical. If PromFuzz++ fails to identify a real functional bug, the corresponding case will be discarded and may directly become a missed detection. In contrast, false positives generated at this stage do not immediately become final reports, because they must still pass the subsequent validation. Next, the critical variable and principal statement extraction stage further examines whether a candidate is supported by concrete variables and statements required for constructing invariant checkers. If no relevant variables or statements can be extracted, the candidate is directly discarded. This mechanism helps filter many false positives propagated from the previous stage, since most false-positive candidates do not contain the variables or statements needed to construct a meaningful functional invariant. In such cases, the LLM usually returns `None`, preventing these candidates from being propagated further. However, if incorrect variables or statements are extracted, the error may be passed to the invariant checker integration stage. Then, at the invariant checker integration stage, such errors usually lead to compilation failures, invalid checkers, or checkers inserted into unreachable locations. These cases are unlikely to introduce final false positives, because they are further filtered by compilation validation and fuzzing. However, if such failures occur on real bug candidates, they will prevent the corresponding bugs from being detected and thus introduce false negatives.

Overall, this analysis shows that most LLM-induced errors in PromFuzz++ are filtered by its layer-by-layer analysis process before final reporting, indicating that the impact of error propagation is largely controlled. The remaining robustness limitations mainly lie in missed analysis during the LLM-driven multi-perspective analysis stage and incorrect invariant checker integration for real bug candidates, which respectively account for two and one of the three false negatives reported in Table 10.

RQ4: PromFuzz++ demonstrates robust functional bug detection performance across repeated runs, with small variations in recall and F1-score. The error propagation analysis further shows that most LLM-induced errors are largely controlled.

4.7 Zero-day Functional Bug Detection

This subsection answers RQ5. 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 12, 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. These bugs are identified in active DeFi projects with an aggregate TVL of approximately $18.2 billion at the time of our evaluation. 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

Just Accepted

Section 5 delves into the practical impacts of these bugs, examining specific cases to illustrate how attackers exploit these bugs.

Table 12. 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

RQ5: PromFuzz++ successfully identifies 30 zero-day functional bugs across six DeFi projects, and 24 of these have been assigned CVE IDs.

5 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.

5.1 Risky First Deposit

As shown in Figure 9, the code contains a risky first deposit bug that can lead to an unfair advantage for the first user and a distorted distribution of 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.

5.2 AMM Price Oracle Manipulation

As shown in Figure 10, 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

Just Accepted

```
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. 9. The Risky First Deposit bug (line 10).

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

```
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. 10. The AMM Price Oracle Manipulation bug (line 13).

Just Accepted

6 Discussion

Although PromFuzz++ demonstrates strong effectiveness in detecting functional bugs, several factors may affect its reliability, generalizability, and practical deployment. This section discusses these factors from seven aspects, including LLM reliability, potential LLM pre-training contamination, generalizability to other languages, practical scalability, bug category coverage, invariant checker insertion, and dependence on dynamic exploration.

LLM Reliability. The use of LLMs may introduce variations in intermediate outputs and final detection results due to their stochastic nature and sensitivity to prompts and inference configurations. To mitigate this threat, we use the same prompts, model, and parameter configurations throughout all experiments. We also conduct a dedicated robustness analysis in Section 4.6 to examine the stability of the LLM-based stages and analyze how errors at different stages affect the final detection results. These measures help reduce experimental variations and improve the reliability of PromFuzz++.

Potential LLM Pre-training Contamination. Some real-world cases in our datasets are collected from public attack reports and audit reports, so we cannot completely rule out possible pre-training exposure. However, PromFuzz++ does not use the LLM as a direct vulnerability detector. Instead, the LLM mainly supports program analysis tasks, such as semantic understanding and extracting bug-relevant variables and statements, while the final vulnerability determination relies on dynamically validated invariant checkers. Therefore, prior exposure alone does not directly determine the detection result, although its potential influence cannot be completely eliminated.

Generalizability to Other Languages. The current implementation and evaluation of PromFuzz++ focus on smart contracts written in Solidity and deployed on Ethereum or other EVM-compatible blockchains. Although the general idea of combining bug-oriented LLM analysis with dynamically validated invariant checkers can be applied to other smart contract languages, PromFuzz++ needs to be adapted to their syntax and execution semantics. Extending PromFuzz++ to other smart contract languages and blockchain platforms is an important direction for future work.

Practical Scalability. The analysis cost of PromFuzz++ grows with contract size and complexity, the number of candidate bug types, and the number of generated invariant checkers. Large-scale projects with complex control flow may require longer LLM inputs and more extensive dynamic exploration, thereby increasing both computational and monetary costs. However, PromFuzz++ is not tied to a specific LLM and can adopt more efficient models as they become available. Future work will explore locally deployed and task-specific fine-tuned models, together with more efficient candidate filtering and contract decomposition strategies, to reduce analysis overhead and improve the performance of PromFuzz++.

Bug Category Coverage. The bug-oriented analysis of PromFuzz++ currently targets four primary categories and ten subcategories of functional bugs. Although these categories cover representative functional bugs in smart contracts, bugs outside the predefined taxonomy or involving complex semantics may remain undetected. Future work will extend the bug taxonomy and develop corresponding prompts and invariant checkers to improve the comprehensiveness of PromFuzz++.

Invariant Checker Insertion. Although our evaluation shows that the LLM can effectively perform invariant checker insertion in most evaluated cases, the current insertion procedure does not incorporate an explicit feedback-driven validation and repair mechanism. Therefore, incorrect or imperfect insertions may still affect the subsequent analysis in some cases. The empirical evaluation of checker insertion mitigates this concern by independently assessing its effectiveness, but it cannot completely eliminate this limitation.

Dependence on Dynamic Exploration. The effectiveness of PromFuzz++ partially depends on the execution coverage achieved by *ItyFuzz*. Since *ItyFuzz* is not specifically designed for functional bug detection, it may fail to reach the contract states required to expose deeply state-dependent bugs. Moreover, its current exploration does not fully utilize contract state information to guide functional bug detection. Future work will investigate

Just Accepted

tighter integration between LLM-based analysis and dynamic analysis to better utilize contract state information and guide the exploration of states relevant to functional bugs.

7 Related work

Existing approaches to smart contract security analysis can be broadly categorized into static analysis, fuzzing-based analysis, symbolic-execution-based analysis, invariant-based analysis, and AI-based analysis.

Static Analysis. Tools like *Slither* [17], *Zeus* [22], *Securify* [41], *Madmax* [18], and *Verisol* [44] 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.

Fuzzing-based Analysis. Tools like *ItyFuzz* [34], *Smartian* [14], *Harvey* [48], *Echidna* [19], and *Contract-Fuzzer* [21] execute smart contracts with generated concrete inputs to explore program behaviors and detect vulnerabilities. *ItyFuzz* uses snapshot-based fuzzing to reduce re-execution overhead and prioritizes interesting program states. *Smartian* combines static analysis for seed generation with lightweight dynamic data-flow feedback to guide fuzzing. *Harvey* employs greybox fuzzing with input prediction and demand-driven exploration of transaction sequences. While fuzzing captures more realistic behaviors by executing smart contracts, it still primarily targets implementation bugs and struggles to effectively detect functional bugs.

Symbolic-execution-based Analysis. Tools like *Mythril* [5], *Manticore* [30], *TeEther* [23], and *Oyente* [29] systematically explore program paths using symbolic inputs and constraint solving. *Oyente* applies symbolic execution directly to EVM bytecode to identify security vulnerabilities. *Manticore* provides a dynamic symbolic execution framework for exploring smart-contract state spaces. *Sailfish* [11] adopts a hybrid approach that first performs lightweight exploration to identify suspicious state interactions and then applies symbolic evaluation to precisely validate candidate state-inconsistency bugs. However, they mainly target predefined vulnerability classes and are less effective at detecting functional bugs.

Invariant-based Analysis. Tools like *VeriSmart* [36], *InvCon* [25], *INVCON+* [26], *Trace2Inv* [13], and *SmartOracle* [37] infer invariants to capture security and correctness properties of smart contracts. *VeriSmart* automatically discovers transaction invariants to support the verification of arithmetic safety properties. *InvCon* dynamically infers likely invariants from historical transactions to recover implicit contract specifications. *SmartOracle* infers fine-grained invariants at the contract, function, and branch levels and uses their violations as vulnerability oracles. After further validation, some inferred invariants may reveal a limited number of functional bugs. However, these approaches do not provide an end-to-end pipeline that directly detects functional bugs.

AI-based Analysis. *SmarTest* [35], *ILF* [20], *xFuzz* [49], and *GNN-based tools* [27, 28, 52] 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* [38] and *SMARTINV* [43] have been used to address functional bugs by analyzing discrepancies between business logic and code. *GPTScan* uses LLM-based semantic analysis to examine candidate functions and extract relevant contract features, followed by static analysis to confirm functional bugs. *SMARTINV* uses a finetuned LLM to infer critical program points and smart contract invariants, which are then checked through formal verification and bounded model checking. In contrast, PromFuzz++ generates targeted executable invariant checkers for candidate functional bugs and validates them through stateful fuzzing. Together, these approaches demonstrate different ways of integrating LLMs with security analysis, opening new avenues for functional bug detection in smart contracts.

Just Accepted

8 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 564 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.

9 Acknowledge

This work was partly supported by the Science Challenge Project under No. TZ2025005, NSFC under No. U2441239, 62472347, and 62472276, the "Pioneer and Leading Goose" R&D Program of Zhejiang under No. 2025C02033 and 2025C01082, and Zhejiang Key Laboratory of Decision Intelligence under No. 2025E10006.

References

[1] 2023. Hack3d: The Web3 Security Report 2023. https://www.certik.com/resources/blog/7BokMhPUgffqEvyvXgHNaq-hack3d-the-web3-security-report-2023.
[2] 2024. Code4rena. https://code4rena.com.
[3] 2024. Cryptocurrency Prices, Market Cap and Charts - Digital Assets. https://www.forbes.com/digital-assets/crypto-prices.
[4] 2024. Immunefi. https://immunefi.com.
[5] 2024. Mythril. https://github.com/ConsenSys/mythril.
[6] 2024. SunWeb3Sec/DeFiHackLabs. https://github.com/SunWeb3Sec/DeFiHackLabs.
[7] 2026. Introduction to Smart Contracts. https://ethereum.org/en/developers/docs/smart-contracts/.
[8] 2026. Solidity Documentation. https://docs.soliditylang.org/.
[9] 2026. Vyper Documentation. https://docs.vyperlang.org/.
[10] 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. 2023. Gemini: A Family of Highly Capable Multimodal Models. *CoRR* abs/2312.11805 (2023). arXiv:2312.11805
[11] Priyanka Bose, Dipanjan Das, Yanju Chen, Yu Feng, Christopher Kruegel, and Giovanni Vigna. 2022. SAILFISH: Vetting Smart Contract State-Inconsistency Bugs in Seconds. In *43rd IEEE Symposium on Security and Privacy, SP 2022*. IEEE, 161–178.
[12] 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. 2020. Language Models are Few-Shot Learners. In *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020*.
[13] Zhiyang Chen, Ye Liu, Sidi Mohamed Beillahi, Yi Li, and Fan Long. 2024. Demystifying Invariant Effectiveness for Securing Smart Contracts. *Proc. ACM Softw. Eng.* 1, FSE (2024), 1772–1795. doi: 10.1145/3660786
[14] Jaeseung Choi, Doyeon Kim, Soomin Kim, Gustavo Grieco, Alex Groce, and Sang Kil Cha. 2021. SMARTIAN: Enhancing Smart Contract Fuzzing with Static and Dynamic Data-Flow Analyses. In *36th IEEE/ACM International Conference on Automated Software Engineering, ASE 2021*. IEEE, 227–239.
[15] Yue Duan, Xin Zhao, Yu Pan, Shucheng Li, Minghao Li, Fengyuan Xu, and Mu Zhang. 2022. Towards Automated Safety Vetting of Smart Contracts in Decentralized Applications. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, CCS 2022, Los Angeles, CA, USA, November 7-11, 2022*, Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi (Eds.). ACM, 921–935.
[16] Thomas Durieux, João F. Ferreira, Rui Abreu, and Pedro Cruz. 2020. Empirical review of automated analysis tools on 47, 587 Ethereum smart contracts. In *ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020*, Gregg Rothermel and Doo-Hwan Bae (Eds.). ACM, 530–541.

Just Accepted

[17] Josselin Feist, Gustavo Grieco, and Alex Groce. 2019. 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*. IEEE / ACM, 8–15.

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

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

[20] Jingxuan He, Mislav Balunovic, Nodar Ambroladze, Petar Tsankov, and Martin T. Vechev. 2019. 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*. ACM, 531–548.

[21] Bo Jiang, Ye Liu, and W. K. Chan. 2018. 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*. ACM, 259–269.

[22] Sukrit Kalra, Seep Goel, Mohan Dhawan, and Subodh Sharma. 2018. ZEUS: Analyzing Safety of Smart Contracts. In *25th Annual Network and Distributed System Security Symposium, NDSS 2018*. The Internet Society.

[23] Johannes Krupp and Christian Rossow. 2018. teEther: Gnawing at Ethereum to Automatically Exploit Smart Contracts. In *27th USENIX Security Symposium, USENIX Security 2018*. USENIX Association, 1317–1333.

[24] Haonan Li, Yu Hao, Yizhuo Zhai, and Zhiyun Qian. 2023. Assisting Static Analysis with Large Language Models: A ChatGPT Experiment. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2023*, Satish Chandra, Kelly Blincoe, and Paolo Tonella (Eds.). ACM, 2107–2111.

[25] Ye Liu and Yi Li. 2022. InvCon: A Dynamic Invariant Detector for Ethereum Smart Contracts. In *37th IEEE/ACM International Conference on Automated Software Engineering, ASE 2022*. ACM, 160:1–160:4.

[26] Ye Liu, Chengxuan Zhang, and Yi Li. 2025. Automated Invariant Generation for Solidity Smart Contracts. *IEEE Trans. Dependable Secur. Comput.* 22, 6 (2025), 7172–7188. doi: 10.1109/TDSC.2025.3592705

[27] Zhenguang Liu, Peng Qian, Xiang Wang, Lei Zhu, Qinming He, and Shouling Ji. 2021. 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*. ijcai.org, 2751–2759.

[28] Zhenguang Liu, Peng Qian, Xiaoyang Wang, Yuan Zhuang, Lin Qiu, and Xun Wang. 2023. Combining Graph Neural Networks With Expert Knowledge for Smart Contract Vulnerability Detection. *IEEE Trans. Knowl. Data Eng.* 35, 2 (2023), 1296–1310.

[29] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. 2016. Making Smart Contracts Smarter. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 254–269.

[30] Mark Mossberg, Felipe Manzano, Eric Hennenfent, Alex Groce, Gustavo Grieco, Josselin Feist, Trent Brunson, and Artem Dinaburg. 2019. Manticore: A User-Friendly Symbolic Execution Framework for Binaries and Smart Contracts. In *34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019*. IEEE, 1186–1189.

[31] 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. 2022. 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*.

[32] Anton Permenev, Dimitar K. Dimitrov, Petar Tsankov, Dana Drachsler-Cohen, and Martin T. Vechev. 2020. VerX: Safety Verification of Smart Contracts. In *2020 IEEE Symposium on Security and Privacy, SP 2020, 2020*. IEEE, 1661–1677.

[33] Ben Prystawski, Michael Li, and Noah D. Goodman. 2023. 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*.

[34] Chaofan Shou, Shangyin Tan, and Koushik Sen. 2023. ItyFuzz: Snapshot-Based Fuzzer for Smart Contract. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2023*. ACM, 322–333.

[35] Sunbeom So, Seongjoon Hong, and Hakjoo Oh. 2021. SmarTest: Effectively Hunting Vulnerable Transaction Sequences in Smart Contracts through Language Model-Guided Symbolic Execution. In *30th USENIX Security Symposium, USENIX Security 2021*. USENIX Association, 1361–1378.

[36] Sunbeom So, Myungho Lee, Jisu Park, Heejo Lee, and Hakjoo Oh. 2020. VERISMART: A Highly Precise Safety Verifier for Ethereum Smart Contracts. In *2020 IEEE Symposium on Security and Privacy, SP 2020*. IEEE, 1678–1694.

[37] Jianzhong Su, Jiachi Chen, Zhiyuan Fang, Xingwei Lin, Yutian Tang, and Zibin Zheng. 2025. SmartOracle: Generating Smart Contract Oracle via Fine-Grained Invariant Detection. *IEEE Trans. Software Eng.* 51, 4 (2025), 947–959. doi: 10.1109/TSE.2025.3534567

[38] Yuqiang Sun, Daoyuan Wu, Yue Xue, Han Liu, Haijun Wang, Zhengzi Xu, Xiaofei Xie, and Yang Liu. 2024. GPTScan: Detecting Logic Vulnerabilities in Smart Contracts by Combining GPT with Program Analysis. In *46th IEEE/ACM International Conference on Software Engineering, ICSE 2024*.

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

Just Accepted

[40] 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. 2023. LLaMA: Open and Efficient Foundation Language Models. In *arXiv preprint arXiv:2302.13971*.

[41] Petar Tsankov, Andrei Marian Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Bünzli, and Martin T. Vechev. 2018. 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*. ACM, 67–82.

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

[43] Sally Junsong Wang, Kexin Pei, and Junfeng Yang. 2024. SMARTINV: Multimodal Learning for Smart Contract Invariant Inference. In *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, 125–125.

[44] Yuepeng Wang, Shuvendu K. Lahiri, Shuo Chen, Rong Pan, Isil Dillig, Cody Born, Immad Naseer, and Kostas Ferles. 2019. 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 (Lecture Notes in Computer Science, Vol. 12031)*. Springer, 87–106.

[45] 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. 2022. Emergent Abilities of Large Language Models. *Trans. Mach. Learn. Res.* 2022 (2022).

[46] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H. Chi, Quoc V. Le, and Denny Zhou. 2022. 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*.

[47] Yuxiang Wei, Chunqiu Steven Xia, and Lingming Zhang. 2023. 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*. ACM, 172–184.

[48] Valentin Wüstholz and Maria Christakis. 2020. 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*. ACM, 1398–1409.

[49] Yinxing Xue, Jiaming Ye, Wei Zhang, Jun Sun, Lei Ma, Haijun Wang, and Jianjun Zhao. 2024. xFuzz: Machine Learning Guided Cross-Contract Fuzzing. *IEEE Trans. Dependable Secur. Comput.* 21, 2 (2024), 515–529.

[50] Zhuo Zhang, Brian Zhang, Wen Xu, and Zhiqiang Lin. 2023. Demystifying Exploitable Bugs in Smart Contracts. In *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023*. IEEE, 615–627.

[51] 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. 2023. 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*. ACM, 5673–5684.

[52] Yuan Zhuang, Zhenguang Liu, Peng Qian, Qi Liu, Xiang Wang, and Qinming He. 2020. Smart Contract Vulnerability Detection using Graph Neural Network. In *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI 2020*. ijcai.org, 3283–3290.

Just Accepted