Identifying Smart Contract Security Issues in Code Snippets from *Stack Overflow*

Jiachi Chen
Sun Yat-sen University
Zhuhai, China
chenjch86@mail.sysu.edu.cn

Chong Chen
Sun Yat-sen University
Zhuhai, China
chench578@mail2.sysu.edu.cn

Jiang Hu
Sun Yat-sen University
Zhuhai, China
hujiang5@mail2.sysu.edu.cn

John Grundy
Monash University
Melbourne, Australia
john.grundy@monash.edu

Yanlin Wang*
Sun Yat-sen University
Zhuhai, China
wangylin36@mail.sysu.edu.cn

Ting Chen
University of Electronic Science and Technology of China
Chengdu, China
brokendragon@uestc.edu.cn

Zibin Zheng
Sun Yat-sen University
Zhuhai, China
zhzibin@mail.sysu.edu.cn

Abstract

Smart contract developers frequently seek solutions to developmental challenges on Q&A platforms such as *Stack Overflow (SO)*. Although community responses often provide viable solutions, the embedded code snippets can also contain hidden vulnerabilities. Integrating such code directly into smart contracts may make them susceptible to malicious attacks. We conducted an online survey and received 74 responses from smart contract developers. The results of this survey indicate that the majority (86.4%) of participants do not sufficiently consider security when reusing *SO* code snippets. Despite the existence of various tools designed to detect vulnerabilities in smart contracts, these tools are typically developed for analyzing fully-completed smart contracts and thus are ineffective for analyzing typical code snippets as found on *SO*. We introduce *SOChecker*, the first tool designed to identify potential vulnerabilities in incomplete *SO* smart contract code snippets. *SOChecker* first leverages a fine-tuned *Llama2* model for code completion, followed by the application of symbolic execution methods for vulnerability detection. Our experimental results, derived from a dataset comprising 897 code snippets collected from smart contract-related *SO* posts, demonstrate that *SOChecker* achieves an F1 score of 68.2%, greatly surpassing GPT-3.5 and GPT-4 (20.9% and 33.2% F1 Scores respectively). Our findings underscore the need to improve the security of code snippets from Q&A websites.

*corresponding author

ISSTA '24, September 16–20, 2024, Vienna, Austria

ACM ISBN 979-8-4007-0612-7/24/09
https://doi.org/10.1145/3650212.3680353

CCS Concepts

• **Software and its engineering → Software testing and debugging**.

Keywords

smart contracts, large language models, program analysis

ACM Reference Format:
Jiachi Chen, Chong Chen, Jiang Hu, John Grundy, Yanlin Wang, Ting Chen, and Zibin Zheng. 2024. Identifying Smart Contract Security Issues in Code Snippets from *Stack Overflow*. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '24), September 16–20, 2024, Vienna, Austria.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3650212.3680353

1 Introduction

In recent years, smart contracts have catalyzed the development of many new applications, such as Non-fungible Tokens (NFTs) [74] and Decentralised Finance (DeFi) [54]. Due to the rapidly evolving of blockchain technology and the limited availability of online resources, developers often turn to Q&A platforms such as *Stack Overflow (SO)* [2] for development guidance. Such Q&A platforms may facilitate knowledge exchange and may help to address the questioner's issue. However, the shared code snippets in answers can also embed hidden vulnerabilities, posing significant security risks when incorporated naively into smart contracts, especially by inexperienced smart contract developers.

Various methods, such as static analysis [24, 40, 56, 62], dynamic analysis [58] and formal verification [53], have been proposed to detect vulnerabilities in smart contracts, these approaches usually require a fully complete and compilable smart contract code. However, conducting such security analyzes directly on incomplete shared smart contract code snippets from *SO* posts presents significant unsolved challenges. Consequently, when analyzing code snippets from *SO* posts, these tools may fail for the majority of cases. Recent studies have demonstrated the promising capabilities of Large Language Models (LLMs) [69] in code-related tasks,

including code completion [22] and code generation [78]. However, research by Chen et al. [10] highlights that the direct use of LLMs, such as ChatGPT-4, for detecting smart contract vulnerabilities has produced very unsatisfactory results, a domain where traditional program analysis techniques excel [11]. The strength of traditional program analysis techniques lies in their ability to enhance the comprehension of complex code structures through abstract data (e.g., control flow graph [6] and data flow graph [17]). This capability is often beyond the reach of LLM. In addition, LLM is susceptible to issues such as hallucinations and randomness [75], which can lead to decreased accuracy in vulnerability detection.

To confirm if smart contract developers use vulnerable code from *SO* code snippets, we ran an online survey and received 74 valid responses. Our survey results show that 88.4% of smart contract practitioners rely on Q&A websites such as *SO* to solve problems encountered during the development process. However, less than 20% of these practitioners then conduct thorough security checks on the code they reuse from these *SO* posts. This suggests that forum-sourced smart contract code snippets indeed present high potential security risks. According to the survey feedback, "lack of support for direct code analysis on *SO*" is the main reason why developers do not apply existing tools to security analysis of code snippets, which highlights the importance of tools like SOChecker.

To address this major real-world issue of incomplete smart contract code snippet security analysis, we introduce SOChecker, a tool that combines the code completion capabilities of LLMs with traditional program analysis methods. SOChecker is the *first* tool specifically designed to analyze fragmented smart contract code on Q&A websites such as *Stack Overflow* and able to detect nine common smart contract vulnerabilities listed in DASP10 [70], e.g., *Reentrancy*, *Access Control*, etc. [11]. SOChecker comprises two key components: a *Code Completer* and a *Vulnerability Detector*. For the former we employ the *Llama2* model [61], fine-tuned with a dataset of the top 1,000 smart contracts with the highest transaction volume from the Ethereum mainnet, to enhance code completion capabilities. While a LLM can successfully complete the semantics of the program, the code it produces may occasionally exhibit syntax issues, such as incompatible *Solidity* versions or missing structural symbols. Hence, we developed scripts for automated version matching and code structure completion to address these issues. After completing SO code snippets into compilable contracts, we use a conventional *Vulnerability Detector* approach for security analysis. Specifically, we first construct a Control Flow Graph (CFG) [6] of the contract. Considering that we only aim to detect vulnerabilities in the original code snippets and code added by the LLM may introduce new vulnerabilities, we implement a program pruning strategy to remove paths generated by LLM code from the CFG. SOChecker is able to detect all nine vulnerabilities categorized by DASP10 [70], a widely recognized smart contract vulnerability list.

To assess the efficacy of SOChecker, we curated a dataset of 897 Solidity code snippets from smart contract-related *SO* posts. In the code completion stage, our fine-tuned model successfully completed 75.5% of code snippets, outperforming the *Llama2* base model, as well as widely used *GPT-3.5-turbo*, and *GPT-4* LLMs. In the vulnerability detection stage, our experimental results show that SOChecker achieved an F1 score of 83.4%, greatly outperforming 10 state-of-the-art smart contract vulnerability detection tools. When applying SOChecker to analyze vulnerabilities in *SO* code snippets directly, it achieves an F1 score of 68.2%, while the scores for *GPT-3.5* and *GPT-4* are only 20.9% and 33.2%, respectively.

In this research, we make the following key contributions:

- We conducted a survey to collect the perspectives of smart contract practitioners on the usage of *SO* code, demonstrating the high potential security risks associated with smart contracty code snippet usage from Q&A websites.
- We introduced SOChecker, the *first* tool that combines code completion capabilities of LLMs with the program analysis methods to analyze smart contract code snippets found on Q&A websites like *Stack Overflow*. SOChecker is able to detect nine common smart contract vulnerabilities.
- We curated a high-quality dataset consisting of 897 Solidity code snippets from smart contract-related posts on *SO* posts. We used this dataset to evaluate SOChecker. The results indicate that the effectiveness of SOChecker is as high as 68.2%, surpassing GPTs and other traditional vulnerability detection tools.
- To promote further research in related fields, we make available our dataset, experimental results, and source code of SOChecker at https://github.com/BugmakerCC/SOChecker [3].

2 Motivation and Background

2.1 Motivating Example

In Figure 1, we show a *SO* post [1] as an example, where the questioner posted a question about a Solidity programming issue. In addition to answering the question, the respondent also provided a code snippet for reference. Although this code may serve the immediate needs of the inquirer – evidenced by the acceptance of the answer – it conceals a *Denial of Service* [73] vulnerability hidden in it. This vulnerability could be exploited by a malicious attacker to disrupt the normal execution of this function, preventing legitimate participants from receiving payment. Even worse, this vulnerable code may also be reused by other developers reading this SO post, if they encounter similar issues. If several people up-rate the post, it may become a popular solution despite the vulnerability.

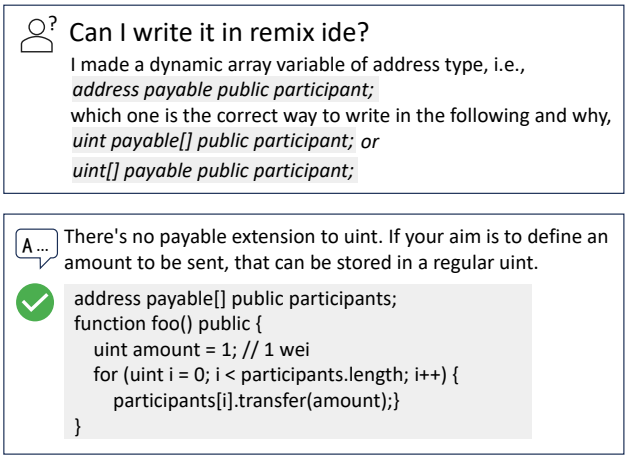

Figure 1: A post on *Stack Overflow* related to *Solidity*.

[1] https://stackoverflow.com/questions/72171101/can-i-write-it-in-remix-ide

Table 1: Top 9 smart contract vulnerabilities in DASP10 and their corresponding descriptions.

Vulnerability	Description
Reentrancy (RE)	Reentrant function calls make a contract to behave in an unexpected way
Access Control (AC)	Failure to use function modifiers or use of tx.origin
Arithmetic Issues (AI)	Integer over/underflows
Unchecked Return Values (URV)	call(), callcode(), delegatecall() or send() fails and it is not checked
Denial of Service (DoS)	The contract is overwhelmed with time-consuming computations
Bad Randomness (BR)	Malicious miner biases the outcome
Front Running (FR)	Two dependent transactions that invoke the same contract are included in one block
Time Manipulation (TM)	The timestamp of the block is manipulated by the miner
Short Address Attack (SAA)	EVM itself accepts incorrectly padded arguments

Detecting vulnerabilities in this code snippet is not easy. Firstly, like most such code snippers, the code in the post is fragmented and uncompilable, which cannot be straightforwardly analyzed using traditional contract vulnerability detection tools. Unlike conventional program analysis tools, LLMs can directly detect vulnerabilities in code snippets without requiring compilable code. However, Chen et al. [10] have shown that LLMs face challenges in directly detecting vulnerabilities in contract code, frequently resulting in false positives. Therefore, a more accurate and precise method is needed to analyze the security of smart contract code snippets.

2.2 DASP10 Smart Contract Vulnerabilities

The DASP10 [70] is a list of common smart contract vulnerabilities and is frequently referenced in academic research [10, 25]. Smartbugs [21], a framework built upon the DASP10 vulnerability classification, integrates multiple vulnerability detection tools and has been employed in numerous subsequent studies [10, 20]. A summary of each vulnerability listed in DASP10 is shown in Table 1. Only the top nine vulnerabilities are discussed, since the 10th vulnerability of DASP10 is "Unknown Unknowns", representing all undiscovered vulnerabilities. While some research [72] has outlined increasingly complex types of smart contract vulnerability, our study focuses primarily on code snippets sourced from *Stack Overflow (SO)*. These snippets are typically brief, straightforward in logic, and generally free of intricate vulnerabilities. Therefore, we adopt DASP10 as the detection standard.

2.3 Large Language Models

A Large Language Model (LLM) is a machine learning model acquired through extensive training on a substantial corpus of text data. This allows it to comprehend and generate natural language and other textual formats proficiently [34, 69]. Throughout their training process, most LLMs acquire extensive code knowledge from code-based training data. This results in high proficiency in code-related tasks, e.g., code generation, comprehension, and summarization[30]. Research indicates a growing inclination among programmers to use LLMs to generate code to aid in work[64]. The code completion ability of LLMs has also garnered recognition [22]. Furthermore, relevant studies have developed LLM-based code translation tools that have shown strong performance [49].

GPT-4 [5] is an LLM developed by *OpenAI* specifically for natural language processing and text generation. As the latest iteration of the *GPT* series, it builds on and refines the technological advances of its predecessor. In particular, surpassing *GPT-3*, *GPT-4* showcases substantial improvements in terms of model size, training data size, and overall performance. This model follows a closed-source approach with a commercial licensing model, which requires users to pay a fee for access rights [47]. Despite the associated cost, *GPT-4* demonstrates exceptional proficiency in a multitude of tasks, making it a widely acclaimed and popular LLM. Recently, *Meta* released their latest open-source large language model, *Llama 2* [61]. Its pre-trained model is trained on 2 trillion tokens and its fine-tuning model has been trained on more than 1 million human annotations [43]. *Llama 2* outperforms other open-source language models on many external benchmarks, including reasoning, coding, proficiency, and knowledge tests [43]. Moreover, *Llama 2* is available for free research and commercial use [43], so we can build datasets specific to a task and fine-tune it based on the model, making the fine-tuned model more capable of handling the task.

2.4 Pre-training, Fine-tuning and Inference

LLMs typically undergo pre-training and fine-tuning during their training process. In the pre-training phase, the model is exposed to extensive text data to acquire linguistic knowledge that includes grammar, context, and semantics [57]. Following completion of pre-training, the model often undergoes fine-tuning to tailor its capabilities to specific tasks, e.g., code generation and summarization [55]. The fine-tuning stage generally employs supervised learning [14], utilizing labeled data for additional training to adjust model parameters and align with the requirements of the targeted tasks.

Inference refers to the process in which a trained model generates output based on input data [4]. This typically occurs when the model has completed training and is prepared to process real-world data. This step is crucial to applying the model to practical problems and tasks. LLMs suffer from instability (i.e., the responses generated each time are different) and hallucinations (i.e., the responses generated contain things that have not appeared in context) during inference due to sensitivity to adversarial samples[68], overfitting [29], and complex context.

3 Real-world *Stack Overflow* Smart Contract Code Snippet Usage

3.1 Motivation

While smart contract codes on *Stack Overflow (SO)* may contain vulnerabilities, it remains unclear whether these codes are actually

utilized by developers in practice. We conducted an online survey specifically targeting real-world smart contract developers to try and determine how often this actually happens. Our survey was designed to gather insights on smart contract developer perspectives and usage of code snippets from *SO*. This includes their criteria for using code snippets, approaches to evaluate and modify code quality and security before integrating them into their projects. Based on the information, we can assess the risks of using such community-contributed code.

3.2 Survey Design

We followed Kitchenham and Pfleeger's instructions [35] for personal opinion surveys and designed an anonymous survey to increase response rates [63]. Our survey was made available in both English and Chinese, since English is the most widely used language and Chinese has the largest number of speakers worldwide. Bi-lingual co-authors carefully reviewed the two versions and guaranteed their consistency. For each question, we made it an optional question to prevent practitioners from not understanding it or being unwilling to answer it. The following provides a brief introduction to our survey questions. For the complete questionnaire, please refer to our online supplementary material [3].

Demographics (Q_{1-4}). We collected the following demographic information to understand the background of respondents, and filter those who might not fully understand our survey.

- Smart contract practitioner? *Yes / No.* (Q_1)
- Main role as a smart contract practitioner. *Development / Testing / Project Management / Research / Other.* (Q_2)
- Experience in years. *Free-text.* (Q_3)
- Current country of residence. *Free-text.* (Q_4)

Access Frequency of *Stack Overflow* (Q_{5-6}). We wanted to ask practitioners to self-assess their familiarity with *SO* (Q_5), and how often they access the Q&A platform (Q_6).

Questioners' Perspective (Q_{7-12}). We wanted to examine how practitioners, acting as questioners, perceive and engage with smart contract codes on *SO*. We first assessed how frequently these practitioners ask questions on *SO* (Q_7), and the types of questions related to smart contracts (such as grammar, security issues, API uses, etc) that most concern them (Q_8). Additionally, we asked whether practitioners used code directly from *SO* (Q_9). For those who have used SO code snippets, we asked about the security analysis they performed prior to code incorporation into their own smart contract programs (Q_{10}). Conversely, for practitioners who have never used *SO* code snippets, we asked for their reasons (Q_{11}). Finally, we askded about the security assessment measures, e.g., code reviews, that practitioners adopt when evaluating code from *SO* (Q_{12}).

Respondents' Behaviour (Q_{13-15}). We asked about how often practitioners respond to others' questions on *SO* (Q_{13}). Subsequently, we asked respondents if they verify the security of any code snippets they add to their answers (Q_{14}), and if so, the methods they use to ensure code security before sharing it (Q_{15}).

Understanding of Smart Contract Vulnerabilities (Q_{16-18}). We showed a set of example smart contract code vulnerabilities as outlined in DASP10 [70], inviting practitioners to explain both their understanding (Q_{16}) and their perception on the importance of identifying these vulnerabilities in *SO* code (Q_{17}). Additionally, we asked practitioners to suggest additional contract vulnerabilities that they consider necessary to detect (Q_{18}).

Usage of tools (Q_{19-21}). We asked practitioners about the use of existing smart contract vulnerability detection tools. First, we investigated how often practitioners utilize these tools in their development process (Q_{19}), and whether they apply these tools for *SO* code (Q_{20}). Then, we asked practitioners' perspectives on the main limitations associated with employing these tools for security analysis of *SO* code (Q_{21}).

Suggestions for Improvement (Q_{22}). Finally, we asked what aspects do practitioners think need improvement when reviewing or using community-shared smart contract code snippets (Q_{22}).

3.3 Survey Validation

We conducted a pilot survey with a small number of practitioners to obtain feedback on whether the questions are clear and easy to understand. The participants included our academic collaborators and partners working in well-known blockchain companies. Based on the feedback, we refined some questions for enhanced clarity without adding or removing any questions. We also polished our translation to further reduce ambiguity between the two language versions of the survey.

3.4 Participant Recruitment

We adopted a non-probabilistic [26] strategy for participant recruitment. Specifically, we conducted a keyword-based search for smart contract repositories on Github, extracted their contributors' emails via the Github REST API [27], and sent the survey to them. Our selection of keywords encompasses a broad spectrum of topics within smart contract technology, e.g., "smart contract", "solidity", and "erc-20". For the complete keyword list, please refer to our online repository [3]. We then sent our survey to a total of 1,416 smart contract practitioners and received 74 valid responses from 19 countries, a reasonable number compared to previous smart contract related survey [9, 67, 77]. We excluded five responses, as they claim to have no development experience in smart contracts. The roles played by respondents in the field of smart contracts are mainly distributed in research (31, 44.9%), development (52, 75.4%), testing (33, 47.8%), project management (12, 17.4%), security audit (37, 53.6%), compliance check (3, 4.3%), training and education (9, 13.0%) and market analysis (2, 2.9%). Their average years of experience are 2.80 (min: 0.2, max: 7.0, median: 2.0, sd: 2.0).

3.5 Results

Access Frequency of *Stack Overflow*. Only 8.1% participants identified themselves as either "Ignorant" or "Not very familiar" with *SO*, suggesting that a majority of 91.9% possess some degree of familiarity with the platform. Notably, the vast majority (94.6%) of practitioners have accessed *SO*, and 77.0% of the participants reported engaging with *SO* at least once a week. These findings highlight the important role of *SO* in the smart contract ecosystem.

Questioners' Perspective. Although 59.4% of our participants infrequently post questions on *SO*, a majority have used code from the site (88.4%). Before using code from *SO*, only 16.4% performed comprehensive security audits using various methods. Over a third perform basic code reviews without employing additional tools or

methods for security audits (36.1%). Many said they understand the code logic but do not specifically assess code security (31.1%). Few (11.6%) of the participants indicated that they do not refer to the code from *SO*, the predominant reason being that the code does not align with the unique requirements of their projects (60.0%). Figure 2 shows the proportion of participants conducting security analysis on code from *SO* in various ways. We categorize the security status of the code into three distinct levels, determined by the rigor of the auditing methods employed. These levels are defined as "unsecured", "basic security" and "advanced security". Most participants favor self-review to identify obvious errors and vulnerabilities (82.6%). In contrast, the use of more professional audit methods, such as specialized security audit tools (20.3%) or consulting professional auditors (10.1%), is significantly less common.

Observation 1: The vast majority (94.6%) of practitioners have accessed *SO*. Although frequent questioning by practitioners on the platform is rare, many (88.4%) seek solutions by browsing through posts made by others.

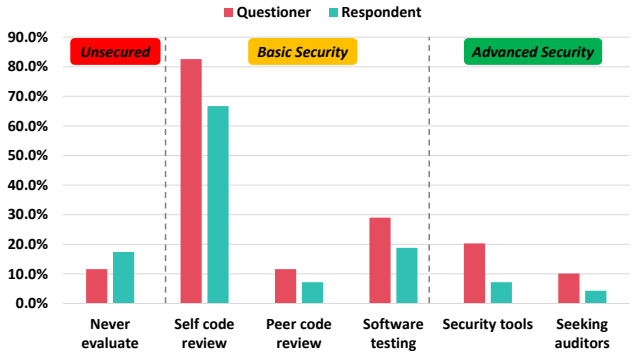

Figure 2: The way for participants to conduct security analysis on the code on *Stack Overflow*.

Respondents' Behaviour. When responding to queries on *SO*, only a very small proportion of respondents (7.1%) consistently verify the security of the code before providing any code-related responses, regardless of the context. A larger group conduct security checks only for complex codes or those involving sensitive functions such as transfers (28.6%). Approximately a quarter (25.7%) check the security of the code in most instances. Almost a quarter (24.3%) of our respondents never evaluate security of their example code before responding to others' questions with it. The most prevalent method employed by respondents for checking code security is through self-review (65.7%). Only a very small fraction utilize professional tools (7.1%) or seeks professional assistance (4.3%) for security analysis. We determined the percentage of participants capable of ensuring advanced security of shared code snippets, and discovered that this group represents merely 13.6% of the total.

Observation 2: Both as questioners and respondents, the majority of participants are only able to ensure basic even lower security for code (86.4%), with a very limited number (13.6%) capable of guaranteeing advanced security.

Familarity with Smart Contract Vulnerabilities. Figure 3 illustrates participants' comprehension of smart contract vulnerabilities and their perceived importance of detecting these vulnerabilities on *SO*. Participants assigned scores ranging from 1 to 5 for each vulnerability, with 1 being the lowest and 5 the highest familarity. Among the vulnerabilities listed in DASP10 [70], participants demonstrated the highest level of understanding regarding *Reentrancy*, with an average score of 4.30. On the contrary, their understanding of *Short Address Attack* was the lowest, averaging at 3.03. Concurrently, *Reentrancy* is also perceived by participants as the most critical vulnerability to detect, receiving an average importance score of 4.54. Beyond the vulnerabilities outlined in DASP10 [70], participants also identified additional concerns, including price manipulation and accuracy issues.

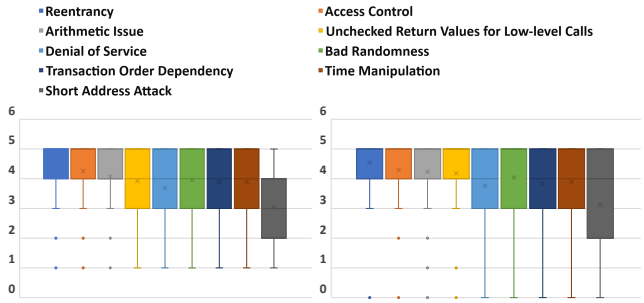

Figure 3: The level of understanding of smart contract vulnerabilities among participants (left) and their belief in the necessity of detecting these vulnerabilities on *SO* (right).

Usage of tools. A significant majority of participants (81.4%) have utilized tools for detecting vulnerabilities in smart contracts. Yet, only a very small 20.0% have applied these tools for the security analysis of code on *SO*. When questioned about their reluctance to use these tools on *SO* code, 44.9% cited "lack of support for direct code analysis on *SO*" as a key factor, while 60.7% pointed to "poor usability of the tools, their complexity, and the high time cost involved" as their primary concerns.

Expectations for *Stack Overflow*. For smart contract codes shared within the community, a significant proportion of our survey participants (72.9%) view security and vulnerability detection as the primary areas in need of improvement or support. Additionally, there is a notable demand for improvements in code quality and clarity, as indicated by 64.3% of the participants.

Observation 3: Despite the availability of numerous tools for smart contract security, only 20.0% of participants reported using these tools on code from *SO*. The predominant reason cited for not using these tools, mentioned by 44.9% of respondents, was the "lack of support for direct code analysis on *SO*".

4 Our SOChecker Approach

4.1 Overview

Figure 4 provides an overview of SOChecker approach. SOChecker comprises two main components: a *Code Completer* and a *Vulnerability Detector*. For the *Code Completer*, we first collect the top 1,000 smart contracts with high transaction volumes from the Ethereum

mainnet. Then, we use them to fine-tune the open-source LLM *llama2-chat-13b* [43], aiming to enhance the model's performance in smart contract code completions. For code snippets that our model cannot complete, we further utilize *GPTs* to generate completions again and then merge the results. Finally, we performed two steps (i.e., structural completion, version adaptation) on the LLM-completed code to increase the number of compilable smart contracts. Our *Vulnerability Detector* utilises two primary steps, code preprocessing and vulnerability detection. During the code preprocessing stage, we compile the completed smart contract code, extract their Abstract Syntax Trees (ASTs), and construct Control Flow Graphs (CFGs). Subsequently, we prune the CFGs based on the original code snippets and ASTs. In the vulnerability detection stage, we developed patterns for nine types of DASP10 vulnerabilities and conduct pattern matching on the pruned CFG to identify potental vulnerabilities. Based on the results of vulnerability detection, we generate a safety report for developers to consult.

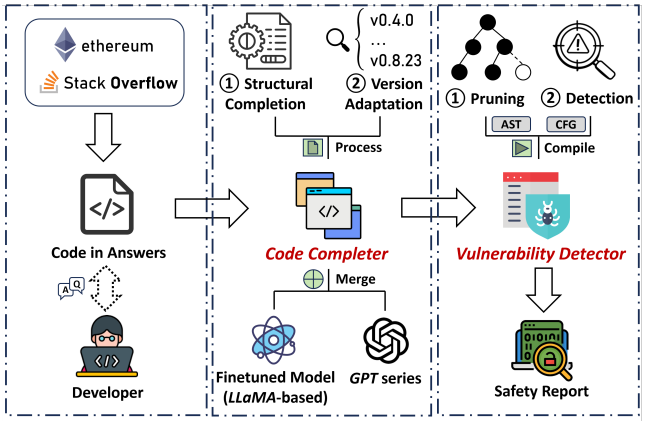

Figure 4: The overall workflow of SOChecker.

4.2 Code Completer

4.2.1 Data Collection. To construct a smart contract dataset for fine-tuning, we first gathered 345,058 open-source smart contracts from the Ethereum mainnet through the GitHub repository *smart-contract-sanctuary* [48] up to March 2023.

To evaluate the code completion performance of the all the candidate models (i.e., *GPTs*, *llama2*, *codellama*), we obtained 4,952 posts related to smart contracts from *Stack Overflow (SO)* after October 2021 [2]. These posts were selected based on the following criteria: 1) The post should have at least one response. 2) The post should include at least one tag of *Solidity*, *Ethereum*, *ERC20*, *ERC721*, or *Contract*. From these 4,952 posts, we refined our dataset by excluding answers for several reasons: 1) Non-code answers were removed, as our analysis focuses find security issues on code snippets. 2) Code was not written in the *Solidity* language was omitted. 3) Brief one-liner code snippets were omitted, as they typically provide limited information and are unlikely to pose significant security risks. Finally, we had a curated dataset comprising 897 code snippets for our subsequent code completion and analysis steps.

[2]The training data for the *GPTs* was up to September 2021 [5]. Selecting posts after this time point helps mitigate the impact of data leakage.

4.2.2 Data Preprocessing & Fine-Tuning. We adopt a fine-tuning process to enhance the model's ability to accurately understand and generate smart contract code. We strategically chose to focus on a subset of the top 1,000 smart contracts with the highest transaction volumes from the initial pool of 345,058 smart contracts for fine-tuning. The average length of these 1,000 smart contracts is 711.03 lines, including comments. Choosing these smart contracts for fine-tuning is to balance the breadth of smart contract applications (e.g., NFTs, DeFi) with the practicality of computational efficiency during fine-tuning. Meanwhile, the selection with the highest transaction volume is based on the assumption that these contracts are more likely to represent real-world scenarios with significant usage and functionalities. Additionally, these smart contracts typically present lower vulnerability risks. From collected data, we found that the majority of code snippets on *SO* are function-level segments. Therefore, function-level code was used as input for LLM fine-tuning, with complete codes serving as the targeted outputs. We construct fine-tuning data by extracting functions from complete contracts.

However, incorporating lengthy smart contracts posed a challenge; their complexity and detailed nature could potentially lead the model to be distracted by the intricacies of the logic itself, thus affecting the effectiveness of the model. To mitigate this issue, we used a method to segment lengthy contracts into shorter parts. Our segmentation process starts with the compilation of each smart contract to obtain its abstract syntax tree (cf. Section 4.3.1), providing insights into the dependency relationships among various subcontracts. For each subcontract, we use its function snippets as the input of a fine-tuning data and include the subcontracts upon which it depends, along with itself, as the output of a fine-tuning data. Figure 5 illustrates a simple example of constructing fine-tuning data. The function *registerUser* on line 1 of "Input" serves as the target objective that requires completion. Given that this function is dependent on other contracts, we include the subcontract *UserRegistration* on line 11 of "Output" containing this function and its dependent contract *Identity* on line 2 of "Output" as the completed code. The task instructions are further detailed in Figure 5. This procedure allows us to construct a dataset of segmented smart contracts for fine-tuning effectively.

```
The following is a snippet of a smart contract's code in Solidity.
Please complete the code to make it a fully functioning, compilable
smart contract in Solidity. Only the complete smart contract code is
needed, do not output other information. The snippet is:
```
Instruction

```
1 function registerUser(address _user) external {
2     require(!isVerified(_user), "User already verified");
3     verifyUser(_user);
4 }
```
Input

```
1 pragma solidity ^0.8.0;
2 contract Identity {
3     mapping(address => bool) private isVerified;
4     function verifyUser(address _user) external {
5         isVerified[_user] = true;
6     }
7     function isVerified(address _user) external view returns (bool) {
8         return isVerified[_user];
9     }
10 }
11 contract UserRegistration is Identity {
12     function registerUser(address _user) external {
13         require(!isVerified(_user), "User already verified");
14         verifyUser(_user);
15     }
16 }
```
Output

Figure 5: An example of constructing fine-tuning data.

4.2.3 Model Selection. In our study, we chose *llama2-chat-13b* [43] and *Codellama-instruct-13b* [51] for fine-tuning with the following reasons: 1) Both of them are open-source, ensuring transparency and accessibility. 2) These LLMs possess a moderate number of parameters [38], resulting in an acceptable computational cost. 3) Both can be fine-tuned to better understand task requirements. 4) Previous research and relevant work [7, 12, 50] have demonstrated their effectiveness in various contexts. After fine-tuning, we compared their performance in the code completion task and then selected the model with better performance for further analysis.

We used the dataset constructed in Section 4.2.1 to fine-tune *Llama2-chat-13b* and *Codellama-instruct-13b*. We employed LoRA technology [31] for model fine-tuning. The training parameters were configured with a rank of 8, alpha of 32, batch size of 256, 3 epochs, and a learning rate of 1e-4. All parameter settings remain default. To assess their performance, we randomly selected 324 smart contracts based on a confidence interval of 10 and a confidence level of 95% [32] from the *smartbugs-wild* dataset [21], which comprises 47,398 smart contracts. Each smart contract's functions were extracted as code snippets for completion with the same method introduced in Section 4.2.2. In particular, these code snippets do not overlap with the fine-tuning dataset. The experimental results reveal that the fine-tuned *llama2-chat-13b* provides 141 compiled smart contracts, whereas the fine-tuned *Codellama-instruct-13b* only provides 48. This indicates that *llama2-chat-13b* serves as a more suitable base model for fine-tuning in the task of smart contract code completion.

4.2.4 Structural Completion. Smart contract programming demands high syntactical precision, including the accurate placement of structural symbols, e.g., ')', '}'. LLMs may struggle with the correct prediction of such symbols due to complex contextual relationships, often generating non-compilable code. To mitigate this issue, we implemented a preprocessing step for the LLM-generated code and developed a script to intelligently insert the missing structural symbols, thus ensuring the syntactical completeness and integrity of the code.

4.2.5 Solidity Version Adaptation. The version declaration of the smart contract is susceptible to errors. We observed instances where the code for certain contracts, despite being correct, failed compilation, with the issue solely attributed to the version declaration of *Solidity*. This challenge arises from the numerous versions of *Solidity* and the subtle differences between each version, making it challenging for LLMs to accurately discern the correct version from their extensive learned knowledge. To address this issue, we disregarded the version numbers generated by LLMs. Instead, we implemented scripts to systematically test and compile smart contracts across all *Solidity* versions, thereby ensuring the compatibility and successful compilation.

4.3 Vulnerability Detector

4.3.1 Code Preprocessing. The complete smart contract code undergoes compilation by the *Solidity* compiler, resulting in the generation of the corresponding Abstract Syntax Tree (AST) [45] and bytecode. The source code information contained within the AST

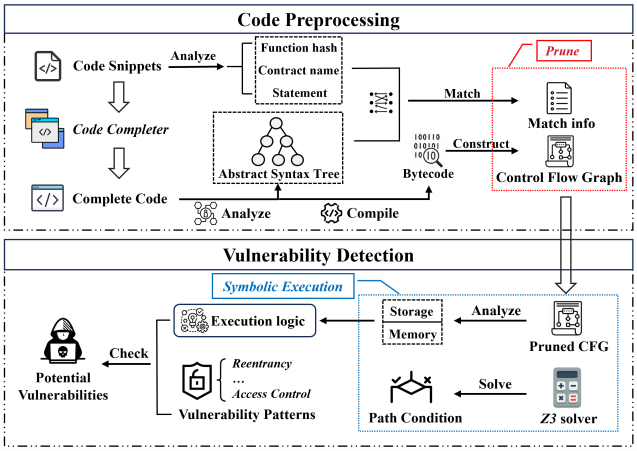

Figure 6: The overall workflow of *Vulnerability Detector*.

can be utilized for subsequent pruning steps. For bytecode, we utilize the API offered by Geth [1] to disassemble it, enabling us to obtain the corresponding opcodes. Subsequently, we segment the opcodes into basic blocks and facilitate block-to-block jumps to finalize the construction of the Control Flow Graph (CFG) [6].

Before performing program analysis, we prune the CFG of the smart contract to concentrate solely on original code fragments. While LLMs are capable of expanding code fragments into full smart contracts, the LLM-generated code might introduce bugs or vulnerabilities not in the original SO code snippet itself. Our pruning approach thus mitigates the influence of LLM-generated code and decreases the probability of false positives. The pruning algorithm is shown in Algorithm 1. We first compile the smart contract and obtain its AST. Then, we extract some key information (e.g., function names and contract names) from the original code snippet through regular expression matching and initialize an empty list *subgraphs* to store the subgraph related to the original code snippet in the AST. Next, we traverse each node in the AST to determine whether the information of that node matches the information in $info$. If so, we extract the subgraph where the node is located and add *subgraph* to *subgraphs*. Finally, we merge and store all subgraphs in *prunedAST*, and extract the corresponding CFG based on the information of each node in *prunedAST*. This will result in a pruned CFG.

It should be noted that pruning may fail for the following two reasons: 1) The original code snippet lacks sufficient information to extract statement-level or higher details from the AST, leading to unsuccessful matching. 2) LLMs incorrectly completed the original code (e.g., changed function names, deleted parts of the snippet source code), resulting in failure to retrieve any information about the original code snippet from the AST.

4.3.2 Vulnerability Detection. After obtaining the pruned control flow graph, we carry out vulnerability detection on the original code snippet. Given that the code snippets found on *SO* are typically simple, it is less likely we will encounter complex vulnerabilities, e.g., a price manipulation attack [36]. Consequently, our analysis concentrates on vulnerabilities as categorized by the Decentralized Application Security Project (DASP) Top 10 [70].

Algorithm 1 Pruning of smart contract snippet

Input: **Contract** *ctr*, **Snippet** *snp*
Output: Pruned *CFG*

```
ast ← GetAST(ctr)
info ← ExtractInfo(snp)
subGraphs ← []
for node in ast do
    if NodeInfo(node) in info then
        subGraph ← ExtractSubGraph(node)
        subGraphs.append(subGraph)
    end if
end for
prunedAST ← SubgraphMerging(subGraphs)
prunedCFG ← ExtractCFG(prunedAST)
return prunedCFG
```

Numerous studies have defined patterns associated with these DASP10 vulnerabilities [24, 40, 56, 62]. Building on their efforts, we have encapsulated these patterns within the program's CFG to facilitate vulnerability detection via symbolic execution. *Z3* SMT solver [18] was used in *Vulnerability Detector*. Taking *Denial of Service (DoS)* as a case in point, our initial step involves analyzing the CFG for loops, which suggests the presence of loops between program blocks. If no loop is present, exit; otherwise, proceed. We iterate through all instructions within blocks of the loop, identifying any resource-intensive operations such as 'CALL' associated with functions like *call* and *transfer* in the code. Such functions typically involve substantial gas consumption. If these instructions are found, we conclude that a *DoS* vulnerability has been detected. For details of each vulnerability pattern please consult our repository [3].

5 SOChecker Evaluation

We conducted a detailed empirical evaluation of SOChecker, focusing on answering three key research questions:

- RQ1: How effective is SOChecker's *Code Completer* in code completion of smart contract snippets?
- RQ2: How effective is SOChecker's *Vulnerability Detector* in vulnerability detection in completed, pruned smart contract code?
- RQ3: How does SOChecker perform when detecting vulnerabilities in real code snippets from *Stack Overflow (SO)*?

For RQ1, we assessed the effectiveness of the *Code Completer* on 897 code snippets collected from *SO* (details see 4.2.1). We evaluated both the compilability and correctness of all completed code, resulting in a LLM-completed smart contract dataset that was accurately completed by LLM. For RQ2, we evaluated the performance of our *Vulnerability Detector* using the LLM-completed smart contract dataset, which is also compatible with traditional vulnerability detection tools due to the compilability of the code in the dataset. This approach enables a fair comparison with other traditional vulnerability detection tools. For RQ3, we conducted a comprehensive evaluation of SOChecker's performance on the entire dataset and compared it with the performance of other LLMs (i.e., GPT-3.5-turbo and GPT-4). By addressing these three research questions, we aim to comprehensively evaluate SOChecker's capabilities in detecting vulnerabilities in real code snippets from *SO*.

5.1 RQ1: Effectiveness of *Code Completer*

We evaluated the effectiveness of *Code Completer* by applying our fine-tuned model to the 897 real code snippets from *SO*, as detailed in Section 4.2.1. When conducting the code completion process with our fine-tuned model, we observed a degree of uncertainty in the model's output, with some code snippets fail to be completed during the first iteration but successfully handled in the subsequent iterations. To fully leverage the model's capabilities, we employed multiple iterations of the code completion process. All the parameters of models (e.g., temperature, decoding strategy) remain default.

Total Compilable Code. As shown in Figure 7, all the models have undergone 13 rounds of iteration, resulting in a continuous increase in the total number of successfully compiled smart contracts, reaching a plateau after 13 iterations. After 13 iterations, our fine-tuned model performs the best, indicating that our fine-tuning is effective for this task. Interestingly, the final performance of *GPT-3.5-turbo* is slightly better than that of *GPT-4*, which we believe is normal because although *GPT-4* is a new version released after *GPT-3.5-turbo*, it may also use updated training data and methods, which can lead to better performance on certain tasks while lowering it on others. In addition, the *GPT* series models showed excellent performance at the beginning, and after the first round of code completion, they provide more than half of the compilable smart contracts. However, as the number of iterations continues to increase, the performance of the *GPT* models decrease significantly. On the contrary, our model's performance has consistently improved, and after 13 iterations, it provides more compilable smart contracts than all other models.

Table 2: The code completion performance of models iterating on datasets.

Models	# Compilable	# Correct	Time	Price
Base	537	442	**4.4h**	-
GPT-3.5-turbo	774	669	7.8h	$8.27
GPT-4	736	649	11.2h	$57.88
Ours	795	677	6.3h	-
Ours+GPT	**889**	**846**	7.1h	$1.32

Quality of Compilable Code. We found that some models produced compilable code but deviated from expected behavior, attributable to design and training characteristics of LLMs. For example, LLMs may automatically repair vulnerabilities when completing the code or may change the logic of the original code. This behavior may stem from exposure to data and rules during LLM's training process, or may result from the model's biased interpretation of tasks or contextual understanding. Consequently, only assessing the volume of compilable code is insufficient – the correctness of the model-completed code is equally important. We manually analyzed the code completed by various models. Table 2 presents an evaluation of this metric. Our *Code Completer* outperforms other LLMs in both the quantity and quality of compilable contracts.

Cost. Table 2 shows the costs of the 4 models on this task. Column 'Time' shows the time required for each model to perform a complete round of code completion. It can be seen that the single round code completion time of *GPT-4* reached 11.2 hours, nearly twice of our model and three times of the base model. Due to the need to pay a fee for each API call [47], the economic cost of GPT

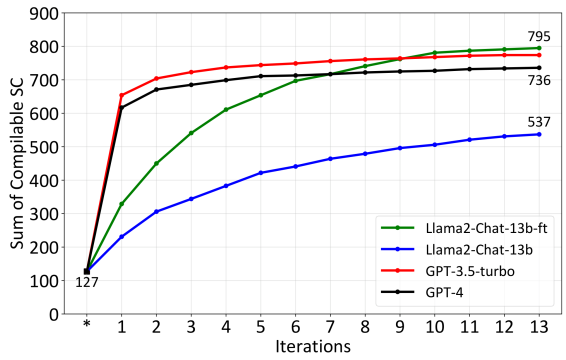

Figure 7: Performance of different models for code completion. The point with a horizontal coordinate of '*' on the graph represents the number of code that could have been directly compiled.

series models also counts. The total cost required for *GPT-4* is the highest, reaching $57.88, followed by *GPT-3.5-turbo* at $8.27. On the contrary, whether it is the open-source *llama2* base model or the model we have fine-tuned, they are all deployed locally and do not require any economic cost. In order to obtain more completed smart contracts, we consider using a compromise solution, which is to perform ablations with the inference results of other models based on the inference results of our *Code Completer*. Finally, we successfully obtained 846 correctly completed code snippets.

We observed the remaining 51 code snippets that cannot be completed by any of the models. There may be the following reasons: 1) inherent flaws within the snippets themselves, such as syntax errors, data type mismatches, and multiple constructors. 2) code complexity also plays a crucial role. This complexity can be attributed to two aspects: the length and clarity of the code. Longer code snippets are more challenging for the model to complete, as they require memorizing and understanding numerous details. Additionally, unclear code snippets, often containing undefined or unconventional variable and function names (e.g., *foobar*, *abcd*), further complicate the model's ability to comprehend and complete the code. While many smart contract codes use traceable naming conventions that aid in logical inference (e.g., *transfer*, *withdraw*), some snippets from *SO* employ non-standard names, making them harder for the model to interpret.

> **Answer to RQ1:** *Code Completer* completed 75.5% of the smart contract snippets correctly, outperforming both its base model and the GPT models, while also offering a lower usage cost compared to the GPT models.

5.2 RQ2: Effectiveness of *Vulnerability Detector*

We assess the efficacy of *Vulnerability Detector* using 846 complete code snippets, as referenced in Section 5.1, which are all correctly completed by the models. Given that nearly all other SOTA tools for vulnerability detection are designed for complete, compiled smart contract code, we can facilitate a fair comparison between our vulnerability detectors and these tools using this subset.

We employed SmartBugs [19], a comprehensive framework that consolidates various smart contract vulnerability detection tools, to execute the evaluations. The selection criteria for these tools were: (1) They target Solidity source code. (2) *Smartbugs* establishes a clear mapping rule between the vulnerability naming of the tool and DASP10 vulnerability naming (Because different tools may employ different naming for the same vulnerability). (3) The tool detects at least one vulnerability listed in the DASP10 [70]. Consequently, we selected the following tools for our study: *Conkas* [65], *Maian* [46], *Mythril* [13], *Osiris* [59], *Oyente* [40], *Securify* [62], *Slither* [24], *Honeybadger* [60] and *Manticore* [44].

Our experiments were carried out on a Windows 11 system equipped with a 12th generation Intel i7 processor, 16GB of RAM, and a 5-minute timeout limit [10] for each tool. Two experienced smart contract researchers independently annotated the presence of vulnerabilities in all snippets of the smart contract. In instances of disagreement, two researchers engaged in discussions to reconcile and unify their conclusions. We used seven metrics to evaluate the experimental results, namely true positive (TP), true negative (TN),

Table 3: Comparison of performance between *Vulnerability Detector* and other tools (*w.a.* F1 denotes weighted average F1 score for all vulnerabilities, and # NUM denotes the number of contracts containing corresponding vulnerabilities).

Vulnerability		RE	AC	AI	URV	DoS	BR	TM
# Num		2	6	10	15	2	8	22
Conkas	# TP	0	-	0	0	-	-	0
	# FP	22	-	17	0	-	-	1
	# FN	2	-	10	15	-	-	22
w.a. F1: 0	F1 %	0	-	0	0	-	-	0
Maian	# TP	-	1	-	-	-	-	-
	# FP	-	2	-	-	-	-	-
	# FN	-	5	-	-	-	-	-
w.a. F1: 22.2%	F1 %	-	22.2	-	-	-	-	-
Mythril	# TP	0	2	0	1	-	-	-
	# FP	6	3	3	0	-	-	-
	# FN	2	4	10	14	-	-	-
w.a. F1: 12.3%	F1 %	0	36.4	0	12.5	-	-	-
Osiris	# TP	0	-	0	-	0	-	0
	# FP	0	-	10	-	0	-	0
	# FN	2	-	10	-	2	-	22
w.a. F1: 0	F1 %	0	-	0	-	0	-	0
Oyente	# TP	0	0	0	-	0	-	0
	# FP	0	0	26	-	0	-	0
	# FN	2	6	10	-	2	-	22
w.a. F1: 0	F1 %	0	0	0	-	0	-	0
Securify	# TP	0	0	-	0	-	-	-
	# FP	0	1	-	0	-	-	-
	# FN	2	6	-	15	-	-	-
w.a. F1: 0	F1 %	0	0	-	0	-	-	-
Slither	# TP	0	1	-	0	0	-	1
	# FP	1	3	-	10	1	-	0
	# FN	2	5	-	15	2	-	21
w.a. F1: 6.6%	F1 %	0	20.0	-	0	0	-	8.7
Smartcheck	# TP	0	0	0	3	1	-	0
	# FP	0	0	3	5	29	-	0
	# FN	2	6	10	12	1	-	22
w.a. F1: 7.1%	F1 %	0	0	0	26.1	6.3	-	0
SOChecker	# TP	1	4	8	9	1	5	20
	# FP	0	0	1	0	0	0	0
	# FN	1	2	2	6	1	3	2
w.a. F1: **83.4%**	F1 %	**66.7**	**80.0**	**84.2**	**75.0**	**66.7**	**76.9**	**95.2**

false positive (FP), false negative (FN), precision, recall and F1 score. TP and TN represent smart contracts with certain vulnerabilities correctly detected by our *Vulnerability Detector* and smart contracts without certain vulnerabilities, respectively. FP and FN indicate that *Vulnerability Detector* has incorrectly detected smart contracts with or without certain vulnerabilities. We calculate precision using formula $P = TP/(TP+FP)$, recall using formula $R = TP/(TP+FN)$, F1 using formula $F1 = 2 * P * R/(P + R)$. For w.a. F1, we calculate it as follows: $w.a.F1 = \frac{\sum_{i=1}^{n} v_i * F1_i}{\sum_{i=1}^{n} v_i}$, where n represents the type of vulnerability and v_i represents the number of vulnerabilities.

Table 3 presents the results of our experiments. Tools *Honeybadger* and *Manticore* are absent from Table 3 as they failed to identify any positive cases within the dataset. Due to the absence of *Front Running* and *Short Address Attack* vulnerabilities in our dataset, we also omitted them in Table 3. Our experimental results reveal that our *Vulnerability Detector* obtains the highest weighted average F1 score of 83.4%, surpassing other SOTA tools across all indicators. A comparative analysis with other tools reveals that our *Vulnerability Detector* not only encompasses all DASP10 [70] vulnerabilities but also exhibits superior detection performance across all categories, underscoring its high efficacy. However, *Vulnerability Detector* still generate some false alarms. We examined each of them individually and found that they were caused by several factors. *Z3* has inherent difficulties in handling complex path conditions, particularly those involving factorial or loop operations, which can easily lead to timeouts and affect the acquisition of critical path information. Additionally, different compiler versions can influence detection results. For instance, *Solidity* v0.8.x includes default integer overflow checks, which differ from earlier versions. *Vulnerability Detector* relies on predefined vulnerability patterns and may not account for these optimizations, leading to discrepancies with actual vulnerabilities.

> **Answer to RQ2:** Our *Vulnerability Detector* achieved an average F1 score of 83.4% on the dataset. Compared to the 10 state-of-the-art tools, it not only identifies the most types of vulnerability listed in DASP10, but also demonstrates the best detection performance for each type of vulnerability.

5.3 RQ3: Effectiveness of SOChecker

We evaluated SOChecker's overall performance with authentic code snippets sourced from *SO*. We executed SOChecker on 897 code snippets collected from *SO*; all experimental settings, such as temperature, maintain the same as them in RQ1 and RQ2.

LLMs can also be directly used for vulnerability detection of smart contract snippets. Consequently, we employed GPT-3.5 and GPT-4 to analyse the same dataset, facilitating a comparative analysis of their performance against SOChecker. Specifically, we designed a prompt informed by others' previous work [10] to obtain non-binary results (i.e., in LLM's response, the presence of vulnerabilities is indicated by "1", while their absence is denoted by "0".). We then gave these results back to *GPT-4* for semantic analysis, so that binary results about the existence of these vulnerabilities can be obtained. In Figure 8, within the vulnerability detection prompt, "[VULS]" denotes the names of all vulnerabilities, "[Input]" specifies the target code subject to detection, and "[CONCLUSION]" represents the detection outcome provided by ChatGPT. For the semantic analysis prompt, "[VULS]" continues to signify the vulnerability name, "[CONCLUSION]" refers to the conclusion derived from prior vulnerability detection, and "[RESULT]" indicates the semantic analysis result delivered by ChatGPT.

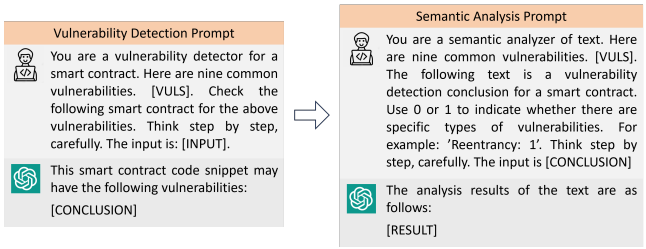

Figure 8: Example of using LLM to detect vulnerabilities.

Table 4 displays the performance of SOChecker and GPT series models in detecting vulnerabilities across 897 real smart contract snippets. The weighted average F1 score achieved by SOChecker is 68.2%, in contrast to GPT-3.5 and GPT-4, which scored 20.9% and 33.2%, respectively. Although GPTs achieving high recall rates for most vulnerabilities, their precision remains low, leading to suboptimal overall performance. This finding aligns with the outcomes of prior research [10]. We observed that SOChecker's recall rate was relatively low. Upon individually analyzing each false negative, we discovered that the majority were attributed to the model either patching the original vulnerabilities or altering the original semantics during code completion. Such issues prove challenging for pruning algorithms to effectively address.

Table 4: Comparison of performance between SOChecker and GPTs.

Vuls	*GPT-3.5*			*GPT-4*			SOChecker		
	P%	*R%*	*F1%*	*P%*	*R%*	*F1%*	*P%*	*R%*	*F1%*
RE	1.8	100	3.5	2.0	100	4.0	100	50.0	**66.7**
AC	0.8	50.0	1.7	1.6	83.3	3.2	100	66.7	**80.0**
AI	7.3	33.3	12.0	8.1	50.0	14.0	88.9	44.4	**59.3**
URV	11.4	65.0	19.4	26.3	75.0	39.0	100	45.0	**62.1**
DoS	1.6	50.0	3.0	1.9	100	3.7	50.0	25.0	**33.3**
BR	17.9	70.0	28.6	30.8	80.0	44.4	83.3	50.0	**62.5**
TM	22.7	63.0	33.3	37.5	77.8	50.6	95.2	74.1	**83.3**
w.a.	13.4	57.5	20.9	23.1	73.6	33.2	92.0	55.2	**68.2**

We tried to execute other vulnerability detection tools discussed in Section 5.2 on this code snippets dataset for comparative analysis. However, as these traditional tools are designed to analyze complete smart contracts, they encountered errors with 770 code snippets presented in fragmentary form. Of the remaining 127 complete contracts, encompassing a total of 18 vulnerabilities, only *Slither* and *Smartcheck* managed to detect 1 and 2 TPs, respectively, while SOChecker identified 14. This outcome suggests that, in the context of fragmented code, our tool demonstrates greater utility compared to conventional tools.

> **Answer to RQ3:** SOChecker achieved an average F1 score of 68.2% on real datasets sourced from *SO*, surpassing both GPT models and other traditional vulnerability detection tools.

6 Threats to Validity

Internal Threats. A potential internal threat in our study is the reliance on specific tags for collecting posts from *SO*. This method may overlook relevant discussions that do not feature the designated tags. However, by selecting widely-used tags such as "Solidity", "Ethereum", "ERC20", "ERC721" and "Contract", we aim to capture a diverse range of smart contract-related content. The extensive volume of posts gathered from these tags helps mitigate this risk and ensures comprehensive coverage for analysis.

External Threats. We designed a survey to assess whether developers had implemented risky *SO* code into their projects. However, we did not directly trace these codes on the Ethereum chain due to the vast amount of contract information available, which made it impractical to complete within our timeframe. Nonetheless, the data gathered from the survey can provide valuable insights into the real-world scenario. Our survey targets smart contract practitioners with diverse backgrounds and levels of experience, allowing them to provide feedback on their use of *SO* code.

7 Related Work

Due to the rise of large language models, some scholars have recently conducted in-depth research on them. Fan et al. [23] studied whether automatic program repair technology can fix the error solutions generated by LLMs in the *LeetCode* competition. Li et al. [37] studied the limitations of LLMs in generating software failure-induced test cases and proposed a differential prompt method to improve effectiveness. Ma et al. [41] evaluated the performance of *ChatGPT* in various subdomains of software engineering. Chen et al. [10] evaluated the performance of *ChatGPT* in detecting vulnerabilities in smart contracts. David et al. [16] studied the detection ability of LLMs such as *Claude* and *GPT* for actual attacks on smart contracts.

Smart contract vulnerability detection has always been a research hotspot in the fields of blockchain and smart contract security. Many work uses static analysis methods to detect potential vulnerabilities in code before contract deployment [8, 24, 40, 56, 58, 62]. Some work has also pointed out the problems with current mainstream tools [79]. Fuzzing is also a commonly used method to detect vulnerabilities in smart contracts [28, 33, 71] In addition, machine learning has also been used in smart contract vulnerability detection tasks in recent years [39, 52]. Our approach to vulnerability detection aligns with established methodologies but offers unique features compared to previous work [8, 24, 40, 56, 58, 62]. Firstly, it possesses the capability to prune programs effectively, rendering it adept at handling the fragmented code commonly found on *SO*. Secondly, it simulates program execution with greater completeness, capturing a wider array of operation codes. Thirdly, given that the majority of the targets are simple contracts, efficiency optimization is not a primary concern, allowing us to incorporate detailed mechanisms like memory and storage mapping.

The code issues on the famous developer forum *Stack Overflow (SO)* have received increasing attention from researchers in recent years. Despite previous efforts [42, 66, 76] to perform security analysis on *SO* code, most of them focus on traditional programming languages (e.g., *C/C++*, *Java*), while *Solidity*, the most popular programming language for smart contracts [15], has received less attention. In addition, these works are mainly completed by manual code inspection, leaving a gap in automated vulnerability detection for Solidity-based smart contracts. Zhang et al. [76] empirically studied the prevalence of the Common Weakness Enumeration (CWE), in code snippets of *C/C++* related answers. Verdi et al. [66] investigated security vulnerabilities in *C++* code snippets on *SO* over a period of 10 years. Meldrum et al. [42] evaluated the quality of *SO* code in various aspects, including reliability and conformance to programming rules, readability, performance and security.

8 Conclusion

We conducted a survey to investigate their usage patterns and perspectives regarding smart contract code snippets on *Stack Overflow* and obtained feedback from 74 smart contract practitioners. Our findings suggest a significant risk associated with the adoption of vulnerable code snippets by developers, potentially compromising the security of the blockchain ecosystem. We wanted to support developers to identify such vulnerabilities within smart contract code snippets. To do this we introduced SOChecker, a novel tool that combines a fine-tuned *Llama2*-based *Code Completer* with a *Vulnerability Detector*. Tested on 897 code snippets, SOChecker demonstrated greatly superior performance over existing GPT-serires LLMs and other program analysis tools.

Acknowledgements

This work is partially supported by fundings from the National Key R&D Program of China (2022YFB2702203), the National Natural Science Foundation of China (62302534, 62332004).

References

[1] 2022. Geth. https://geth.ethereum.org/docs.
[2] 2023. *StackOverflow*. https://stackoverflow.com/
[3] 2024. SOChecker. https://github.com/BugmakerCC/SOChecker
[4] Kamel Abdelouahab, Maxime Pelcat, Jocelyn Serot, and François Berry. 2018. Accelerating CNN inference on FPGAs: A survey. *arXiv preprint arXiv:1806.01683* (2018).
[5] Open AI. 2023. GPT-4. https://platform.openai.com/docs/models/gpt-4.
[6] Frances E Allen. 1970. Control flow analysis. *ACM Sigplan Notices* 5, 7 (1970), 1–19.
[7] Kamel Alrashedy. 2023. Language Models are Better Bug Detector Through Code-Pair Classification. *arXiv preprint arXiv:2311.07957* (2023).
[8] Priyanka Bose, Dipanjan Das, Yanju Chen, Yu Feng, Christopher Kruegel, and Giovanni Vigna. 2022. Sailfish: Vetting smart contract state-inconsistency bugs in seconds. In *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 161–178.
[9] Amiangshu Bosu, Anindya Iqbal, Rifat Shahriyar, and Partha Chakraborty. 2019. Understanding the motivations, challenges and needs of blockchain software developers: A survey. *Empirical Software Engineering* 24, 4 (2019), 2636–2673.
[10] Chong Chen, Jianzhong Su, Jiachi Chen, Yanlin Wang, Tingting Bi, Yanli Wang, Xingwei Lin, Ting Chen, and Zibin Zheng. 2023. When ChatGPT Meets Smart Contract Vulnerability Detection: How Far Are We? *arXiv preprint arXiv:2309.05520* (2023).
[11] Jiachi Chen, Xin Xia, David Lo, John Grundy, Xiapu Luo, and Ting Chen. 2021. Defectchecker: Automated smart contract defect detection by analyzing evm bytecode. *IEEE Transactions on Software Engineering* 48, 7 (2021), 2189–2207.
[12] Davide Cifarelli, Leonardo Boiardi, Alessandro Puppo, and Leon Jovanovic. 2023. Safurai-Csharp: Harnessing Synthetic Data to improve language-specific Code LLM. *arXiv preprint arXiv:2311.03243* (2023).
[13] ConsenSys. 2021. Consensys/mythril: Security analysis tool for evm bytecode. https://github.com/ConsenSys/mythril
[14] Pádraig Cunningham, Matthieu Cord, and Sarah Jane Delany. 2008. Supervised learning. In *Machine learning techniques for multimedia: case studies on organization and retrieval*. Springer, 21–49.
[15] Chris Dannen. 2017. *Introducing Ethereum and solidity*. Vol. 1. Springer.
[16] Isaac David, Liyi Zhou, Kaihua Qin, Dawn Song, Lorenzo Cavallaro, and Arthur Gervais. 2023. Do you still need a manual smart contract audit? arXiv:2306.12338 [cs.CR]

[17] Alan L. Davis and Robert M. Keller. 1982. Data flow program graphs. *Computer* 15, 02 (1982), 26–41.

[18] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 337–340.

[19] Monika di Angelo, Thomas Durieux, João F. Ferreira, and Gernot Salzer. 2023. SmartBugs 2.0: An Execution Framework for Weakness Detection in Ethereum Smart Contracts. In *Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering (ASE 2023)*. to appear.

[20] Monika di Angelo, Thomas Durieux, João F. Ferreira, and Gernot Salzer. 2023), note=to appear. Evolution of Automated Weakness Detection in Ethereum Bytecode: a Comprehensive Study. *Empirical Software Engineering* (2023), note=to appear).

[21] Thomas Durieux, João F. Ferreira, Rui Abreu, and Pedro Cruz. 2020. Empirical Review of Automated Analysis Tools on 47,587 Ethereum Smart Contracts. In *Proceedings of the ACM/IEEE 42nd International conference on software engineering*. 530–541.

[22] Aryaz Eghbali and Michael Pradel. 2024. De-Hallucinator: Iterative Grounding for LLM-Based Code Completion. *arXiv preprint arXiv:2401.01701* (2024).

[23] Zhiyu Fan, Xiang Gao, Martin Mirchev, Abhik Roychoudhury, and Shin Hwei Tan. 2023. Automated repair of programs from large language models. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 1469–1481.

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

[25] João F Ferreira, Pedro Cruz, Thomas Durieux, and Rui Abreu. 2020. SmartBugs: A Framework to Analyze Solidity Smart Contracts. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. 1349–1352.

[26] Manuela Rozalia Gabor et al. 2007. Types of non-probabilistic sampling used in marketing research.„Snowball" sampling. *Management & Marketing-Bucharest* 3 (2007), 80–90.

[27] GitHub. 2023. GitHub REST API documentation. https://docs.github.com/en/rest. Retrieved December 15, 2023.

[28] Gustavo Grieco, Will Song, and Artur Cygan. 2020. Echidna: Effective, usable, and fast fuzzing for smart contracts. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 787–801. https://agroce.github.io/issta20.pdf

[29] Douglas M Hawkins. 2004. The problem of overfitting. *Journal of chemical information and computer sciences* 44, 1 (2004), 1–12.

[30] Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John Grundy, and Haoyu Wang. 2024. Large Language Models for Software Engineering: A Systematic Literature Review. arXiv:2308.10620 [cs.SE]

[31] Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. 2021. Lora: Low-rank adaptation of large language models. *arXiv preprint arXiv:2106.09685* (2021).

[32] Confidence interval Calculator. 2023. Sample Size Calculator. https://www.surveysystem.com/sscalc.htm.

[33] Bo Jiang, Ye Liu, and Wing Kwong Chan. 2018. Contractfuzzer: Fuzzing smart contracts for vulnerability detection. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. 259–269.

[34] Enkelejda Kasneci, Kathrin Seßler, Stefan Küchemann, Maria Bannert, Daryna Dementieva, Frank Fischer, Urs Gasser, Georg Groh, Stephan Günnemann, Eyke Hüllermeier, et al. 2023. ChatGPT for good? On opportunities and challenges of large language models for education. *Learning and individual differences* 103 (2023), 102274.

[35] Barbara A Kitchenham and Shari L Pfleeger. 2008. Personal opinion surveys. In *Guide to advanced empirical software engineering*. Springer, 63–92.

[36] Queping Kong, Jiachi Chen, Yanlin Wang, Zigui Jiang, and Zibin Zheng. 2023. DeFiTainter: Detecting Price Manipulation Vulnerabilities in DeFi Protocols. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 1144–1156.

[37] T. Li, W. Zong, Y. Wang, H. Tian, Y. Wang, S. Cheung, and J. Kramer. 2023. Finding Failure-Inducing Test Cases with ChatGPT. In *IEEE*.

[38] Yuhui Li, Fangyun Wei, Chao Zhang, and Hongyang Zhang. 2024. Eagle: Speculative sampling requires rethinking feature uncertainty. *arXiv preprint arXiv:2401.15077* (2024).

[39] Jian-Wei Liao, Tsung-Ta Tsai, Chia-Kang He, and Chin-Wei Tien. 2019. Soliaudit: Smart contract vulnerability assessment based on machine learning and fuzz testing. In *2019 Sixth International Conference on Internet of Things: Systems, Management and Security (IOTSMS)*. IEEE, 458–465.

[40] 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*. 254–269.

[41] Wei Ma, Shangqing Liu, Wenhan Wang, Qiang Hu, Ye Liu, Cen Zhang, Liming Nie, and Yang Liu. 2023. The Scope of ChatGPT in Software Engineering: A Thorough Investigation. arXiv:2305.12138 [cs.SE]

[42] Sarah Meldrum, Sherlock A Licorish, Caitlin A Owen, and Bastin Tony Roy Savarimuthu. 2020. Understanding stack overflow code quality: A recommendation of caution. *Science of Computer Programming* 199 (2020), 102516.

[43] Meta. 2023. Llama 2. https://ai.meta.com/llama.

[44] 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 *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 1186–1189. https://doi.org/10.1109/ASE.2019.00133

[45] Iulian Neamtiu, Jeffrey S Foster, and Michael Hicks. 2005. Understanding source code evolution using abstract syntax tree matching. In *Proceedings of the 2005 international workshop on Mining software repositories*. 1–5.

[46] Ivica Nikolić, Aashish Kolluri, Ilya Sergey, Prateek Saxena, and Aquinas Hobor. 2018. Finding the greedy, prodigal, and suicidal contracts at scale. In *Proceedings of the 34th annual computer security applications conference*. 653–663.

[47] OpenAI. 2023. GPT Api Pricing. https://openai.com/pricing.

[48] Martin Ortner and Shayan Eskandari. 2022. Smart Contract Sanctuary. https://github.com/tintinweb/smart-contract-sanctuary.

[49] Jialing Pan, Adrien Sadé, Jin Kim, Eric Soriano, Guillem Sole, and Sylvain Flamant. 2023. SteloCoder: a Decoder-Only LLM for Multi-Language to Python Code Translation. *arXiv preprint arXiv:2310.15539* (2023).

[50] Konstantinos I Roumeliotis, Nikolaos D Tselikas, and Dimitrios K Nasiopoulos. 2023. Llama 2: Early Adopters' Utilization of Meta's New Open-Source Pretrained Model.

[51] Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950* (2023).

[52] Christoph Sendner, Huili Chen, Hossein Fereidooni, Lukas Petzi, Jan König, Jasper Stang, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, and Farinaz Koushanfar. 2023. Smarter Contracts: Detecting Vulnerabilities in Smart Contracts with Deep Transfer Learning.. In *NDSS*.

[53] 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)*. IEEE, 1678–1694.

[54] Jianzhong Su, Xingwei Lin, Zhiyuan Fang, Zhirong Zhu, Jiachi Chen, Zibin Zheng, Wei Lv, and Jiashui Wang. 2023. DeFiWarder: Protecting DeFi Apps from Token Leaking Vulnerabilities. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 1664–1675.

[55] Chi Sun, Xipeng Qiu, Yige Xu, and Xuanjing Huang. 2019. How to fine-tune bert for text classification?. In *Chinese Computational Linguistics: 18th China National Conference, CCL 2019, Kunming, China, October 18–20, 2019, Proceedings 18*. Springer, 194–206.

[56] Sergei Tikhomirov, Ekaterina Voskresenskaya, Ivan Ivanitskiy, Ramil Takhaviev, Evgeny Marchenko, and Yaroslav Alexandrov. 2018. Smartcheck: Static analysis of ethereum smart contracts. In *Proceedings of the 1st international workshop on emerging trends in software engineering for blockchain*. 9–16.

[57] Kushal Tirumala, Daniel Simig, Armen Aghajanyan, and Ari S Morcos. 2023. D4: Improving llm pretraining via document de-duplication and diversification. *arXiv preprint arXiv:2308.12284* (2023).

[58] Christof Ferreira Torres, Antonio Ken Iannillo, Arthur Gervais, and Radu State. 2021. Confuzzius: A data dependency-aware hybrid fuzzer for smart contracts. In *2021 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 103–119.

[59] Christof Ferreira Torres, Julian Schütte, and Radu State. 2018. Osiris: Hunting for integer bugs in ethereum smart contracts. In *Proceedings of the 34th annual computer security applications conference*. 664–676.

[60] Christof Ferreira Torres, Mathis Steichen, et al. 2019. The art of the scam: Demystifying honeypots in ethereum smart contracts. In *28th USENIX Security Symposium (USENIX Security 19)*. 1591–1607.

[61] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. 2023. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288* (2023).

[62] Petar Tsankov, Andrei Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Buenzli, and Martin Vechev. 2018. Securify: Practical security analysis of smart contracts. In *Proceedings of the 2018 ACM SIGSAC conference on computer and communications security*. 67–82.

[63] Pradeep K Tyagi. 1989. The effects of appeals, anonymity, and feedback on mail survey response patterns from salespeople. *Journal of the Academy of Marketing Science* 17 (1989), 235–241.

[64] Priyan Vaithilingam, Tianyi Zhang, and Elena L Glassman. 2022. Expectation vs. experience: Evaluating the usability of code generation tools powered by large language models. In *Chi conference on human factors in computing systems extended abstracts*. 1–7.

[65] Nuno Veloso. 2021. Conkas. https://github.com/nveloso/conkas.

[66] Morteza Verdi, Ashkan Sami, Jafar Akhondali, Foutse Khomh, Gias Uddin, and Alireza Karami Motlagh. 2020. An empirical study of c++ vulnerabilities in crowd-sourced code examples. *IEEE Transactions on Software Engineering* 48, 5 (2020), 1497–1514.

[67] Zhiyuan Wan, Xin Xia, David Lo, Jiachi Chen, Xiapu Luo, and Xiaohu Yang. 2021. Smart contract security: A practitioners' perspective. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 1410–1422.
[68] Yixiang Wang, Jiqiang Liu, Jelena Mišić, Vojislav B Mišić, Shaohua Lv, and Xiaolin Chang. 2019. Assessing optimizer impact on DNN model sensitivity to adversarial examples. *IEEE Access* 7 (2019), 152766–152776.
[69] Jason Wei, Yi Tay, Rishi Bommasani, Colin Raffel, Barret Zoph, Sebastian Borgeaud, Dani Yogatama, Maarten Bosma, Denny Zhou, Donald Metzler, et al. 2022. Emergent abilities of large language models. *arXiv preprint arXiv:2206.07682* (2022).
[70] David Wong and Mason Hemmel. 2018. Decentralized Application Security Project Top 10 of 2018. https://dasp.co/index.html.
[71] Valentin W"ustholz and Maria Christakis. 2020. Harvey: A greybox fuzzer for smart contracts. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 1398–1409. https://doi.org/10.1145/3368089.3417064
[72] XBlock. 2023. Smart Contract Defects. https://xblock.pro/#/article/3.
[73] XBlock. 2023. Smart Contract Defects-Denial of Service. http://xblock.pro/#/article/49.
[74] Shuo Yang, Jiachi Chen, and Zibin Zheng. 2023. Definition and Detection of Defects in NFT Smart Contracts. *arXiv preprint arXiv:2305.15829* (2023).
[75] Jia-Yu Yao, Kun-Peng Ning, Zhen-Hui Liu, Mu-Nan Ning, and Li Yuan. 2023. Llm lies: Hallucinations are not bugs, but features as adversarial examples. *arXiv preprint arXiv:2310.01469* (2023).
[76] Haoxiang Zhang, Shaowei Wang, Heng Li, Tse-Hsun Chen, and Ahmed E Hassan. 2021. A study of c/c++ code weaknesses on stack overflow. *IEEE Transactions on Software Engineering* 48, 7 (2021), 2359–2375.
[77] Jiashuo Zhang, Jiachi Chen, Zhiyuan Wan, Ting Chen, Jianbo Gao, and Zhong Chen. 2023. When Contracts Meets Crypto: Exploring Developers' Struggles with Ethereum Cryptographic APIs. *arXiv preprint arXiv:2312.09685* (2023).
[78] Shun Zhang, Zhenfang Chen, Yikang Shen, Mingyu Ding, Joshua B Tenenbaum, and Chuang Gan. 2023. Planning with large language models for code generation. *arXiv preprint arXiv:2303.05510* (2023).
[79] Zibin Zheng, Neng Zhang, Jianzhong Su, Zhijie Zhong, Mingxi Ye, and Jiachi Chen. 2023. Turn the Rudder: A Beacon of Reentrancy Detection for Smart Contracts on Ethereum. *arXiv preprint arXiv:2303.13770* (2023).

Received 2024-04-12; accepted 2024-07-03