

Combining GPT and Code-Based Similarity Checking for Effective Smart Contract Vulnerability Detection

Jango Zhang
Chongqing University
202314021008@stu.cqu.edu.cn

Abstract—With the rapid growth of blockchain technology, smart contracts are now crucial to Decentralized Finance (DeFi) applications. Effective vulnerability detection is vital for securing these contracts against hackers and enhancing the accuracy and efficiency of security audits. In this paper, we present SimilarGPT, a unique vulnerability identification tool for smart contract, which combines Generative Pre-trained Transformer (GPT) models with Code-based similarity checking methods.

The main concept of the SimilarGPT tool is to measure the similarity between the code under inspection and the secure code from third-party libraries. To identify potential vulnerabilities, we connect the semantic understanding capability of large language models (LLMs) with Code-based similarity checking techniques. We propose optimizing the detection sequence using topological ordering to enhance logical coherence and reduce false positives during detection. Through analysis of code reuse patterns in smart contracts, we compile and process extensive third-party library code to establish a comprehensive reference codebase. Then, we utilize LLM to conduct an in-depth analysis of similar codes to identify and explain potential vulnerabilities in the codes. The experimental findings indicate that SimilarGPT excels in detecting vulnerabilities in smart contracts, particularly in missed detections and minimizing false positives.

Index Terms—Smart Contracts, Vulnerability Detection, Code Similarity, LLM, DeFi

I. INTRODUCTION

Since the origin of Ethereum [1], smart contracts have emerged as a fundamental element of blockchain technologies. Their immutable, transparent, and open-source characteristics have established the foundational Decentralized Finance (DeFi) application framework. Given that the DeFi ecology encompasses a multitude of cryptocurrencies [2], it is imperative to identify and address security vulnerabilities in smart contracts. As reported by Defillama Hacks, hacking incidents have resulted in approximately \$9.06 billion in damages through November 2024 [3]. This scenario presents a substantial risk to the security of the entire DeFi ecosystem and user assets. Smart contract vulnerabilities primarily stem from design flaws and coding

errors within decentralized applications, with logical inconsistencies representing a key attack vector for malicious hackers [4]. Existing analysis tools [5]–[8] primarily focus on detecting vulnerabilities that follow predictable static patterns in control and data flows, such as reentrancy issues [9] and integer overflow vulnerabilities [10]. However, these conventional program analysis approaches have shown limited effectiveness in practice [11].

The emergence of generative large language models [12], [13] recently has revealed their significant advantages for auditing smart contracts. Recent investigations [14]–[18] have demonstrated that large language models (LLMs) are promising in auditing smart contracts. Current research in LLM-based vulnerability detection primarily examines smart contracts at the contract or function level, using structured prompts as LLM inputs. The effectiveness of using LLMs for smart contract vulnerability identification is optimized through a two-phase approach that separates the initial detection process from the subsequent analysis of vulnerability root causes. However, this approach heavily relies on Large Language Models’ inherent reasoning abilities and their pre-training knowledge base for detecting vulnerabilities in smart contracts [16]. This challenge can be mitigated by improving the model’s reasoning skills, like fine-tuning. Yet, fine-tuning has limited scalability, risks overfitting to specific datasets, and cannot update knowledge in real-time.

Additionally, research by [16] revealed that simply incorporating vulnerability knowledge into LLMs is insufficient for improving their reasoning capabilities. The study demonstrated that LLMs can effectively reasoning for vulnerability detection when the vulnerability knowledge is carefully structured and systematically integrated.

Recent research [19]–[21] has demonstrated that in the development of Defi, there is a considerable amount of code reuse due to developing costs, code security, and developer habits, and other reasons [19]. This trend is particularly prominent in the development of Solidity smart contracts. The study [21] analyzed over 350,000 Solidity smart contracts to investigate the present state of smart contract composition and code reuse. The study indicated that more than 80% of subcontracts came from external sources, with Node Package Manager(NPM) as

the largest external source, accounting for more than 72% of external subcontracts and less than 17.06% of subcontracts being self-developed. This conclusion emphasizes the dependency on third-party libraries and frameworks for smart contract development. This shows that smart contract development is significantly dependent on third-party package on npm.

Moreover, [21] discovered that approximately 50% of these self-developed subcontracts have fewer than 10% unique functions. Code reuse, including self-developed subcontracts, is common at the function level. This scenario may represent that when confronted with complex functional requirements; developers prefer to reuse existing snippets of code rather than writing them from scratch to save development time and potential security concerns. The report further indicates that despite Solidity's built-in *import* statement for subcontracts, a significant majority (over 56%) of duplicated subcontracts are sourced from NPM package rather than direct imports, indicating a certain amount of uncertainty on dependency management [21].

Similarity checking for vulnerability detection have been developed in response to the extremely high rate of code reuse on Ethereum [22]–[24]. The technique involves analyzing code patterns and structures to identify potential security vulnerabilities, such as insecure mathematical operations and access control vulnerabilities, through comprehensive code comparison techniques, including data flow and control flow analysis, syntactic examination, and various pattern matching methodologies for smart contract verification [24]. Similarity checking for vulnerability detection in contracts face significant limitations in capturing comprehensive contract characteristics - including syntactic elements, semantic meaning, and functional behaviors. This inadequate representation often leads to substantial detection gaps and incorrect vulnerability identifications [24].

To this end, we present SimilarGPT, the first tool that combines GPT and Code-based similarity checking (CBSC) to find coding flaws in smart contracts. Given the significant code reuse problem in Ethereum [21], we use a deep learning model to vectorize the "correct" code (i.e., generic third-party code with no vulnerabilities) and the code to be detected. By measuring the similarity, we can use the "correct" code from related third-party libraries as a reference and feed it into LLMs with the code to be detected for detailed semantic-based detection. This strategy is motivated by the belief that "in school, students with average grades and love to learn will tend to compare their answers with those of high-scores students after completing their after-school homework in order to find out the shortcomings of their solutions".

However, LLMs could generate the illusion that "although this function itself is not vulnerable, we believe that this function is vulnerable because other functions it calls may be vulnerable, which affects the security of

this function". To overcome this issue, SimilarGPT uses a topology-based sequence of function calls, effectively mitigating the impact of the LLM illusion.

To obtain high-quality data for use as similar code, We refer to [21] to collect samples and use our own data improvement methods to filter out negative examples. We collected third-party packages as a dataset, by using [21] as reference. After processing, this dataset consists of the top 150 most frequently used npm and GitHub package, with 35705 files and 357050 functions in total. In the following steps, we polished each function by removing comments, whitespace, indentation, and line breaks before filtering it using hash matching. After those processing procedures, we have 83,321 selected functions. This strategy not only increases data quality, but it also provides us with a dependable reference codebase, which serves as a solid foundation for future code similarity detection and vulnerability detection.

RoadMap. the rest of this paper is organized as follows. We first introduce the related background in II, and then present the detailed design of SimilarGPT in III. Then, we show the experimental setup and results in IV. After that, we discuss the related work and limitations in V and VI, respectively. Finally, VII summarizes the paper.

II. BACKGROUND

A. Smart Contracts and Their Vulnerabilities

Smart contracts enable decentralized finance (Defi) [25] in blockchain transactions, eliminating the need for middleman. According to DeFiLlama [2], the total locked-in value on the three major blockchain platforms - Ether, Solana, and Tron - has reached \$74 billion by November 2024. Smart contract vulnerabilities can lead to property loss, as seen in the *TheDao* event, which cost around \$150 million. However, owing to the characteristic of the blockchain, The contract cannot be modified once deployed on the blockchain, leaving any vulnerabilities open to potential exploitation. [26].

B. LLM-driven Formal Vulnerability Detection of Smart Contracts

Generative pre-trained transformer (GPT) models, such as GPT-4 [12], are extensive language models (LLMs) developed using vast datasets. These models possess the ability to learn from text, comprehend and analyze source code, and perform zero-shot learning [27], enabling them to detect security vulnerabilities in code without needing specific examples.

Despite its promise for code auditing, multiple studies have demonstrated that GPT models cannot replace human auditors. David et al. (2023) shows that inputting the entire project code into a GPT model is both expensive and difficult for achieving accurate detection outcomes. Sun et al. [28] studied the effect of different components (e.g., function calls, external knowledge) on vulnerability detection for LLM. Even under optimal conditions,

such as improving GPT-4’s vulnerability knowledge with Retrieval Augmentation Generation (RAG), the accuracy in the context of LLM’s vulnerability detection paradigm remains below 30% when both the decision (i.e., correctly determining the presence of a vulnerability in the code) and the argument (i.e., correctly pointing out the type of vulnerability) are correct.

C. similarity-based code detection

Code-based similarity checking (CBSC) techniques identify potential vulnerabilities by analyzing and comparing the specific code implementations of different smart contracts. The core idea of this technique is to perform code pattern matching for similar structures using various methods, such as data flow/control flow analysis [22], semantics analysis [24], and symbolic execution [23]. The goal is to identify the potential presence of similar code fragments that may contain known vulnerabilities [24]. For instance, certain code patterns (e.g., insecure mathematical operations and access control vulnerabilities) may be repeated in multiple contracts.

Nevertheless, it is challenging to capture the syntactic, semantic, and functional information of contracts through the abstracted characterization of contracts using CBSC techniques. Therefore, it is likely to result in many false positives and underreporting [24].

III. DESIGN OF SIMILARGPT

The overall design of SimilarGPT and the primary challenges of LLM in detecting smart contract vulnerabilities are depicted in III-A. The three critical components of SimilarGPT are then introduced in III-B, III-C, and III-D, respectively.

A. overview and challenges

The entire framework of SimilarGPT is illustrated in Fig. 1. The green box represents the LLM-based agent, while the blue box represents the data processing. The smart contract code is first preprocessed by breaking it into distinct functions, ensuring each data segment to be evaluated is a separate function. The detection order of these functions is established through topological ordering based on their calling relationship. Subsequently, the detector is provided with similar code from third-party package to conduct a similarity checking for the code. The Socrates section receives the detection result for subsequent evaluation.

Challenge. Despite the concise architecture of SimilarGPT in Fig. 1, it is challenging to conduct effective smart contract auditing, which involves enhancing the detection rate while minimizing the impact of LLMs’ illusions. Several challenges were encountered during the design and implementation of SimilarGPT, as detailed below:

- *How can SimilarGPT effectively detect smart contract vulnerabilities?* While some research has demonstrated the potential of LLMs for smart contract

vulnerability detection, most existing tools cannot combat the increasingly difficult hacking event on the blockchain due to the lack of well-designed knowledge and the rapid iteration of new smart contract vulnerability. This work proposes a GPT-driven Code-based similarity checking for vulnerability detection for smart contract. We will first present this in III-B.

- *How can SimilarGPT effectively handle contextual information?* The previous paper [16] explored how various factors affect LLMs’ ability to detect vulnerabilities in smart contracts, including the impact of model hyperparameter configurations [29] and the integration of external knowledge sources [30]. According to [16], simply providing context info may not always help LLMs’ reasoning about vulnerabilities. It may also cause diversions, preventing LLMs from correctly discovering vulnerabilities. So, how to effectively convey vulnerability-related context will significantly LLMs’ capacity to detect vulnerabilities. We combine a sequence of vulnerability detection functions based on topological ordering into SimilarGPT. It is used to deal with the large model’s contextual information processing during the smart contract detection procedure. This will be given in III-C.
- *How to collect a high quality dataset for code comparison?* Acquiring a high-quality dataset remains a critical foundation, both for conventional vulnerability detection methods that rely on code similarity and for our SimilarGPT approach. We refer to [21]’s method to obtain the dataset and use our own data filtering method to find out the possible negative samples in the dataset. This will be III-D
- *How Can We Reduce False Positive Rate in Large Model Vulnerability Detection?* False positives rate play a critical role in vulnerability detection. However, the hallucination of LLMs [31] constantly leads to false positives for vulnerabilities. As a result, understanding how to reduce the impact of the LLMs illusion becomes critical. We will use recent developments in the Multi-agent Framework in LLMs to connect this topic. Specific details will be presented in III-E.

B. Code-based similarity checking

Fig. 2 shows the first code sample from [4], which focuses on vulnerabilities that machines cannot audit. [4] demonstrates that in Fig. 2, The problem described is difficult to identify, having passed through several rounds of manual auditing and tool evaluation without being discovered. This is because it needs to know the meaning of `_allowances`, the purpose of the `transferFrom` function, and the business model. Fortunately, an ethical hacker discovered the vulnerability and reported it to ImmuneFi [32], a web3 vulnerability bounty site. The project side of Redacted Cartel rewarded the hacker with around \$560,000.

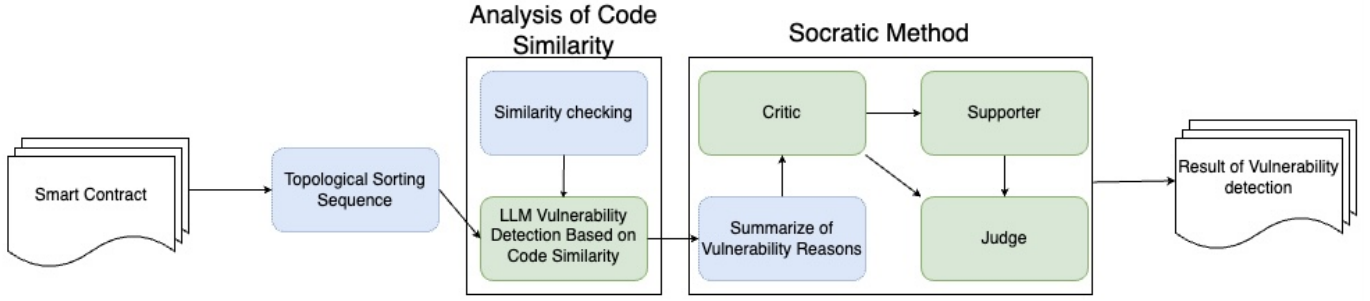


Fig. 1: An overview of SimilarGPT, green blocks indicating GPT works and green blocks suggesting code similar analysis.

```

1  function transferFrom(address from, address to,
   uint256 amount) onlyAuthorizedOperators
   external {
2      _transfer(from, to, amount);
3      _approve(from, msg.sender, allowance(from,
   to)amount);
4      return true;
5  }

```

Fig. 2: The Redacted Cartel exploit

```

1  function transferFrom(address from, address to,
   uint256 amount) external{
2      _transfer(from, to, amount);
3      _approve(from, _msgSender(), _allowances
   [from][_msgSender()]-amount);
4      return true;
5  }

```

Fig. 3: transferFrom function in Openzeppelin’s ERC20 contract

However, the code was probably a clone from the Openzeppelin package. The *transferFrom* function for openzeppelin’s older versions of ERC20 is shown in Fig. 3, and the two functions are largely similar, differing primarily in their modifiers¹. However, the vulnerability can be discovered easily by means of similarity checking. However, simply using CBSC for vulnerability detection makes it hard to capture contracts’ syntactic, semantic, and functional information.

Inspired by this, We use the similar correct code as a reference and use GPT attempt to identify potential vulnerabilities by comparing the differences between the correct third-party package’s code and the code to be identified. The difficulty of understanding function semantics of traditional CBSC is overcome by GPT.

¹Similar to [4], we simplify the actual code to clarify it.

C. Topological Ordering-based Vulnerability Detection Sequencing

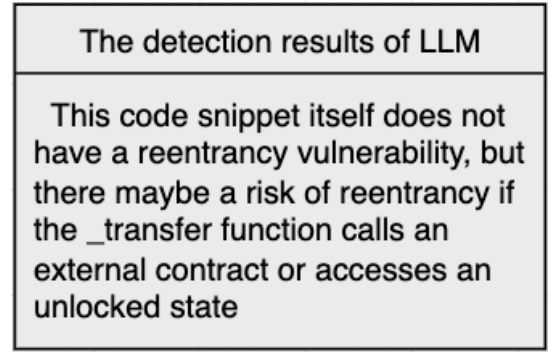


Fig. 4: Detection results of the transferFrom function in Fig. 3

In this paper, Fig. 4 depicts the outcome of using the code in Fig. 3 as a direct input to GPT-4 for vulnerability detection. When conducting vulnerability assessments of Large Language Models (LLMs), we frequently encounter scenarios similar to those illustrated in Fig. 4. The function in Fig. 3 has been audited by some specialists. Thus, we can presume that it has no vulnerability. Similarly, no apparent vulnerabilities were found when LLM examined the transferFrom function. However, it indicated that other functions called by the transferFrom function, such as the function `_transfer` in Fig. 3, are vulnerable. In the end it was determined that this case was considered vulnerable. This is particularly prevalent in the detection of reentrant vulnerabilities. Predictably, there are likely two causes for this predicament. 1. Insufficient context information. 2. Incoherent reasoning resulting from the hallucination of LLMs. We focus on solving the first problem.

It is evident that if LLM thinks that function B, called by function A, has a vulnerability, we can test function B first. Then, we can test function B first and then function A. In the same way, if function B calls function C, then we test function C first. Then, there will always be a function at the bottom of the list which does not call any

other function. Following this, a directed acyclic graph of function calls exists in function calls. Function A is called function B, and A depends on B. So, at the time of detection, B was detected first. In this way, we can perform function-based smart contract detection in the sequence of topological ordering [33]. Of course, some tools can also be used [34] for call graph construction.

Let $G = (V, E)$ be a directed acyclic graph where V represents the set of functions and modifiers in a smart contract, and E represents the calling relationships between them. For any two vertices $v_i, v_j \in V$, an edge $(v_i, v_j) \in E$ indicates that function v_i calls function v_j .

Given that vulnerabilities in called functions can affect their callers, we propose a systematic detection approach based on topological ordering. Let $f : V \rightarrow V$ be a calling relationship where $f(v_i) = v_j$ denotes that v_i calls v_j . Then:

- Let $S = \{v_1, v_2, \dots, v_n\}$ be the set of all functions and modifiers in the contract.
- For each vertex $v_i \in S$, we define:
 - 1) $C(v_i) = \{v_j \in S \mid f(v_i) = v_j\}$ as the set of functions called by v_i
 - 2) For each $v_j \in C(v_i)$, we add edge (v_j, v_i) to E
- Let $\tau : V \rightarrow \mathbb{N}$ be a topological ordering of G such that for every directed edge $(v_i, v_j) \in E$, $\tau(v_i) < \tau(v_j)$

This topological ordering τ provides the optimal sequence for vulnerability detection, ensuring that called functions are always analyzed before their callers. For any $v_i, v_j \in V$, if v_i calls v_j , then $\tau(v_j) < \tau(v_i)$, guaranteeing a bottom-up analysis approach.

As for some possible cases, such as rings in the sequence of function calls, we plan to optimize those scenarios in future work.

D. High-quality Data Collection and Filtering

Collecting code from third-party libraries. In order to employ similarity code for checking, obtaining an extensive amount of code from third-party packages is necessary. We have collected the most frequently used npm and GitHub packages in recent years [19]–[21]. For our data collection methodology, we refer to research from [21], which analyzed over 350,000 smart contract source codes gathered from Etherscan, encompassing both Ethereum mainnet and Goerli testnet deployments between January 2021 and January 2023. A catalog of third-party packages was subsequently generated from the *import* statements of the collated contracts. The 150 most frequently used third-party libraries will be our third-party library code for reference. The utilization of each third-party library between January 2021 and January 2023 was determined through *import* statements and clone detection methods.

For these third-party library codes, we implemented the following approach. Collect the various versions of third-party libraries; for example, as of November 2024, the

npm package of "@openzeppelin/contracts" [35] has 87 distinct versions, which equate to 87 gzip package files. For each third-party library we collect all the different versions of its gzip package. After collecting 150 distinct versions of third-party libraries, we gathered around **10,000** gzip package files. We collect all of the *.sol* files in those gzip packages. Then, we have gathered around **46918** *.sol* files. Moreover, we extract each function in the file and calculate its hash; this allows us to filter out most of the identical code, particularly for multiple versions of the same third-party library with just minor code differences. This way, the vulnerable code that has been modified will be stored. There are **766,505** functions before the hash match and only **35,709** functions following the hash-based filtering.

After hash matching, we use the *all-MiniLM-L6-v2* [36] model based on *sentence-transformers* [37] to convert the code into vectors. *all-MiniLM-L6-v2* model maps sentences and paragraphs to a 384-dimensional dense vector space and can be used for tasks like clustering or semantic search. The model is based on pre-trained *nreimers/MiniLM-L6-H384-uncased* model and fine-tuned in on a 1B sentence pairs dataset. The *all-MiniLM-L6-v2* model is one of the best models based on Sentence Transformers [38]. Compared to other models, [39]–[41] exhibits higher efficiency, and the encoding pace is also fast.

Similarity calculation. Refer to [24], we compute the Euclidean distance to assess code similarity. Specifically, we define the semantic distance and similarity between two code fragments C_1 and C_2 , as well as their related code embeddings e_1 and e_2 , as:

$$\text{Distance}(C_1, C_2) = \frac{\text{Euclidean}(e_1, e_2)}{\|e_1\| + \|e_2\|}$$

Similarity is defined as follows:

$$\text{Similarity}(C_1, C_2) = 1 - \text{Distance}(C_1, C_2)$$

Two code snippets C_i and C_j are considered to be clones if their similarity scores above a certain similarity threshold δ . Based on our observations, 0.65 can be utilized as a similarity threshold. Above 0.65, we may consider the two codes just similar; below 0.65, the two codes maybe different in semantically, structurally, or functionally distinct.

In this method, we can group the code into three categories: code with similarity 1, code with similarity less than 1 but greater than 0.65, and code with similarity less than 0.65.

- 1) *similarity = 1*. As expected, the code with a similarity of 1 is cloned by the developer from third-party libraries such as openzeppelin [35]. we will directly determine if there is a vulnerability in the code.
- 2) *0.65 < similarity < 1*. For codes with similarity less than 1 and more than 0.65, which is the main focus of our research, we incorporate those similar codes into the GPT as part of the prompt.

- 3) *similarity* < 0.65. For codes with a similarity of less than 0.65, no knowledge augmentation is performed, and they are inserted into the prompt as input to the GPT.

Filter vulnerability code. In the case that these third-party libraries contain potential vulnerability code. We employ the current vulnerability datasets for smart contract that have been collected from frequently used public datasets, including defihacks [42], slowmist [43], and github’s issues [44], according to [16]. For the current *top 150 third-party libraries*, we attempt to gather the historical vulnerabilities that have occurred in those third-party libraries. Subsequently, we refer to the GPT-4-based method outlined in [16] to extract relevant vulnerability knowledge from the vulnerability descriptions. We attempt to annotate the corresponding original code by labeling it as a function with a vulnerability after carefully verifying the vulnerability’s existence. Nevertheless, we do not abandon such vulnerable functions; they are employed as illustrations of vulnerabilities for subsequent testing. This is precisely the function of traditional code similarity-based vulnerability detection.

E. Socratic method

Previous research [15], [16], [18] discovered that vulnerability detection based on LLMs generates a substantial number of false positives, especially when the context is not given correctly [16]. Several issues contribute to these false positives, including knowledge mismatch and poor LLM reasoning ability. We are primarily focus with decreasing the false positives caused by LLMs illusions.

As LLM research advances, there is an increasing recognition of Prompt Engineering’s ability to dispel LLMs’ misconceptions. The Socratic method of debate [45]–[48] is very attractive. Asking and responding to questions, starting with generalized beliefs and then testing their internal consistency through rebuttals [47], helps us get closer to the truth, reducing the illusions that may be present in large-scale models and thus improving the accuracy of the models’ judgments. In particular, we established three LLM roles: Critic, Supporter, and Judge [47]. The following is the flow of their interactions:

- **Detector** is a critical component to performing effective smart contract auditing. We feed the code to be detected and the similar code into Detector, and the vulnerabilities are identified by recognizing the differences in implementation between the two pieces of code.
- **Critic & Supporter.** In order to determine the validity of the vulnerability reason during the detecting phase, we use the Socratic method to assess the validity of the output from the detecting phase. Specifically, Critic refutes the cause of the vulnerability given by Detector, while Supporter further evaluates Critic’s output to debate and determine the most appropriate cause of the vulnerability.

- **Judge** is responsible for synthesizing the vulnerability cause, the vulnerability cause arguments given by Critic and Supporter. And, it independently gives its own judgment.

In our future work we consider using multiple rounds of iterations to optimize the Socratic method. [46] [46]

IV. EVALUATION

A. Experimental Setup

In this section, we will provide a summary of some key implementation details about SimilarGPT

GPT model setups. We use the GPT-4-turbo model from OpenAI for SimilarGPT. Our work indicates that GPT-4-turbo could be cost-effective while offering enough inference capacity. The model hyper-parameters are maintained at their default values, with TopP set to 1, presence penalty set to 0, and frequency penalty set to 0. Nonetheless, It should be mentioned that to lessen the impact of the GPT’s output’s unpredictability, we set the temperatures of the three roles-Critic, Supporter, and Judge—to 0. To retain some creativity, the detector’s temperature is set to 0.8, which gives LLM several possibilities to produce some creative results [16]. Furthermore, each agent interaction occurs in a fresh session, ensuring independence between conversations and preventing any potential interference from previous responses.

Datasets. To accurately assess the detection capability of SimilarGPT and the efficacy of CBSC. We employ real-world vulnerabilities and smart contract audit reports from reputable industry companies as datasets. We collect two datasets separately. The first dataset comprises vulnerability data from *Defihack* [42], a well-known DeFi Hacks dataset, and *CVE* (Common Vulnerabilities and Exposures) [49]. We collect 13 typical vulnerability data from the Defihack and CVE vulnerability collections. We have included a variety of types of vulnerabilities, such as overflow, access control, bad randomness, price manipulation, and logic vulnerability, to ensure that the vulnerability data is comprehensive. Those vulnerabilities encompass a range of ages, from 18 to 24. The second dataset is sourced from *Solodit* [50]. This popular smart contract auditing website is specifically designed to facilitate the auditing of Web3 projects, with a particular emphasis on smart contract security. SimilarGPT will be assessed by collecting 67 vulnerable functions from *Solodit* as positive samples and 71 negative samples (i.e., non-vulnerable code). The specific gathering methods are similar to those employed to address vulnerable code from third-party libraries.

Research question. Our proposed method focuses on two primary issues: enhancing recall and lowering false positive rate. Thus, we developed a series of experiments to address the following research questions(RQs):

- 1) **RQ1:** How effective is SimilarGPT at detecting vulnerabilities? How does it compare to traditional vul-

nerability detection methods, including those based on LLM?

- 2) **RQ2:** How helpful is the Socratic method for improving SimilarGPT’s precision rate?
- 3) **RQ3:** How effective is SimilarGPT’s Code-based similarity checking in improving the recall of SimilarGPT?

B. RQ1 - Vulnerability Detection

We examined SimilarGPT’s detection of *real-world vulnerabilities*. These *real vulnerabilities* once led to over \$1,000,000 losses in defi. The settings for SimilarGPT are essentially kept as specified in IV-A, i.e., the parameters are kept as defaults, with only the detector role set to a temperature of 0.8 and the Critic, Supporter and Judge roles set to a temperature of 0. Table I displays the performance of SimilarGPT on 13 *real-world vulnerabilities*. SimilarGPT is compared to *Slither* [6], a popular static analysis tool for detecting vulnerabilities, and *Mythril* [5], a symbolic execution tool. In addition, *Gptlens* [15] is an adversarial framework for vulnerability detection in smart contracts based on the Large Language Model (LLM), aiming to overcome the accuracy and recall concerns of existing LLM tools in vulnerability detection.

Contracts	Description	SimilarGPT	GptLens	Slither	Mythril
Ragnarok	access control	✓	✓	✗	✗
Nimbus Platform	miscalculation	✓	✗	✗	✗
LuckeyTiger	bad randomness	✓	✓	✓	✗
ShadowFi	access control	✓	✗	✗	✗
Grim Finance	reentrancy	✗	✗	✗	✗
Bad Guys by RPF	logic error	✗	✗	✗	✗
Uerii	access control	✓	✓	✗	✗
GPU	logic error	✗	✗	✗	✗
ZongZi	price manipulation	✗	✗	✗	✗
Hopelend	overflow	✓	✗	✗	✗
Coinlancer	access control	✓	✓	✗	✗
Virgo_ZodiacToken	logic error	✓	✗	✗	✗
Lancer	overflow	✗	✗	✗	✓

TABLE I: Vulnerability detection results for 13 real-world vulnerabilities.

In Table I, the first column lists the vulnerability name, followed by the corresponding vulnerability description. The remaining columns provide the detection results for each tool. Table I shows that SimilarGPT outperforms all the tools, with SimilarGPT detecting 8 out of 13 vulnerabilities, followed by *Gptlens* with 4 vulnerabilities, *Slither* detecting only one vulnerability related to bad-randomness, and *Mythril* detecting one vulnerability related to overflow. In those undetected vulnerabilities, the *Lancer* and *ZongZi* contracts were due to that they didn’t provide enough contextual information, including the solidity version and calling function information. The vulnerabilities in the *Bad Guys by RPF* contract were identified due to the absence of modifiers applied to its parameters. Yet Critic and Judge continue to argue that there is no vulnerabilities. The *GPU* weakness was not

identified as a perceived risk, and our think that this may be a result of the knowledge update time of large models. The knowledge of GPT-4-turbo is updated until 2023.

And the capabilities of SimilarGPT can be improved in real time by adding third-party package updates and open-source vulnerability code from different auditing platforms.

Defi-fork-bugs. The vulnerability found in the *Uranium* [51] contract led to losses of about \$50 million. This issue stemmed from a slightly modified function originally from *uniswap v2*. A coding error introduced the bug, causing the significant financial loss. In fact, we can avoid this bug entirely. As early as April 2021, a identical vulnerability had impacted *Nimbus* [52]. If we’re serious about learning this lesson. But even in the 2023, *Swapos* [53] contract experienced the same coding error, although the identical vulnerability had already occurred two years ago. In this time, *Swapos*’s vulnerability resulted in losses of almost \$468,000. Nevertheless, this incident was completely avoidable, and the security weakness that led to it should never have existed in the first place. And, this is where SimilarGPT’s effectiveness may manifest.

Answer for RQ1: SimilarGPT excels in detecting *real-world vulnerabilities*, identifying a broader range of vulnerabilities more effectively compared to other tools. This demonstrates SimilarGPT’s practicality and efficiency in smart contract security analysis.

C. RQ2 - Socratic Debate Framework

The two primary components of the SimilarGPT are vulnerability identification and false positive filtering. While part of CBSC identifies vulnerabilities, the *Socratic method* is responsible for mitigating the impact of the LLM’s hallucination. This section uses the *Solodit* dataset that was previously mentioned in IV-A.

The *Socratic method* [45], [47] is a type of debate method that involves asking and answering questions, beginning with universal beliefs and examining their internal coherence through rebuttals. Here, we attempt to demonstrate the *Socratic method*’s effectiveness in filtering false positives. We refer to the study [15], [18] as a comparison. Specifically, we construct two frameworks. 1) a conventional one-stage framework; 2) a two-stage framework. In the one-stage framework [18], both vulnerability detection and interpretation are performed by the same agent; however, in the latter two-stage framework [14], [15], former agent is responsible for detecting the vulnerability, while the latter is responsible for determining the validity of the vulnerability interpretation. The prompts in two of these frameworks are similar to SimilarGPT, and the same code similarity-based detection mechanism is employed.

In the results shown in Fig. 5, our approach outperforms the traditional model in all four main performance metrics. The fact is that the three frameworks do not differ much in recall rate. Even the single-agent framework performs

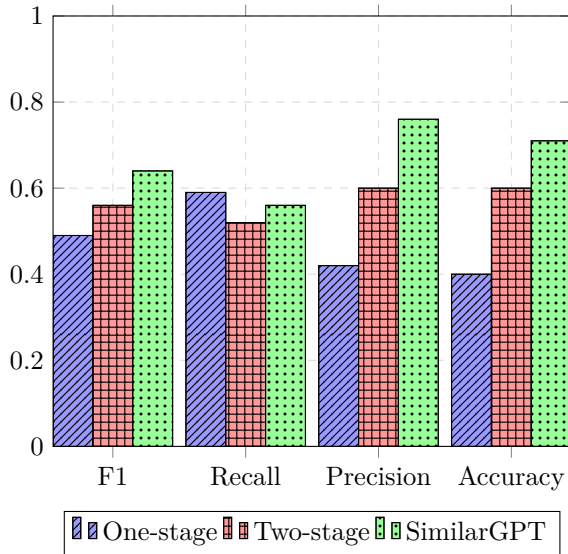


Fig. 5: Comparing SimilarGPT with the traditional integration models.

better on recall. However, when further analyzed, we find that their frameworks do not perform well on negative samples, resulting in the highest false positive rate, reaching 57% false positive rate, for the one-stage. Observing the experimental results suggests this issue is primarily due to hallucinations in LLMs.

Answer for RQ2:the application of the Socratic Debate Framework in SimilarGPT reduced the false positive rate to 12%, compared to 57% in single-agent frameworks. This demonstrates the framework’s effectiveness in refining the detection process by mitigating the influence of hallucinations in large language models.

D. RQ3 - Code-based Similarity Checking

In this section, we try to answer question for *how effective CBSC is in improving the recall rate*.

The dataset for the experiment is the *Solodit dataset*, which contains 138 samples, of which 67 are positive examples and 71 are negative examples. This dataset is also used in the ablation experiment of the Socratic method.

We have implemented modifications to the framework. In the detector section, we have eliminated the CBSC method and instead used the code to be detected as input to the detector. Additionally, the default settings are maintained for the model parameters, except for the temperature option.

	TP	TN	FP	FN	Sum
with SimilarChecking	38	63	12	30	143
without SimilarChecking	20	61	10	47	138

TABLE II: detecting results before and after SimilarChecking.

Table II shows that in the *Solodit dataset*, SimilarGPT discovered 38 true positives (TPs) and produced 12 false positives (FPs). Without a CBSC method, there are only 20 TPs, which is roughly half of the former. However, we highlight that in the absence of CBSC method. The precision rate is 66%. There isn’t much of a difference between these examples and the ones with CBSC method. A plausible assumption is that CBSC method has no substantial influence on precision rates.

Answer for RQ3:the use of Code-based similarity checking in SimilarGPT raised the true positive detections from 20 to 38, highlighting its effectiveness in improving the recall rate and thereby enhancing the tool’s overall accuracy in identifying vulnerabilities.

V. RELATED WORK

Vulnerability Detection. In recent years, vulnerability detection has received a lot of attention, especially given the rapid development of blockchain and smart contract technologies. Early research concentrated on approaches like static analysis, dynamic analysis, and formal verification. And for those static analysis tools, such as *Slither* [6], *Oyente* [8], and others [54]–[56], identify possible vulnerabilities by comparing the code’s syntax and control flow to preset rules. However, such approaches have limitations in dealing with complex logic, dynamic behavior, and low-level vulnerabilities (e.g., re-entry attacks), making it difficult to detect complex logic errors and resulting in too many false positives. Dynamic analysis, like Fuzzy testing *Confuzzius* [7], *Sfuzz* [57], and other tools [58] by automatically generating inputs to test the behavior of the system, as well as symbolic execution *Manticore* *citemanticore*, *Mythril* [5], [59]–[61]. Formal verification [62], [63] ensures the correctness of the code through mathematical proofs. While providing a high degree of accuracy, it is usually applicable to smaller or relatively simple systems, and the verification process is complex and time-consuming.

Based on the observation that the code reuse rate is high on Ethereum [19]–[21], vulnerability detection methods based on code similarity have been properly developed. Liu [23] et al, developed a semantic clone detection tool for Smart Contracts in Ethereum. The method captures essential semantic elements from symbolic transaction execution and converts them into vectors for similarity evaluations. To address the constraints of vulnerability detection, they offer a semantic-aware security auditing approach [64] that evaluates vulnerabilities through N-gram language modeling and static semantic labeling. In addition, [65] enhanced vulnerability detection accuracy by depicting syntactic and semantic aspects of contracts using the birthmark. Huang [22] et al, analyzed data flow and control flow using key instructions and the Graph2Vector tool for similarity. Pierro [66], Chen [67], and Gao [68] evaluated contract similarity and detected vulnerabilities

by analyzing the edit distance of Abstract Syntax Trees (AST), node type hash sequences, and normalization of variables and constant values, respectively. These studies collectively advanced the development of smart contract security analysis.

LLM-based vulnerability detection. Researchers have made extensive use of Large Language Models (LLMs) for vulnerability detection, including Ullah [69], Fu [70], Thapa [71], David [72], Alqarni [73], Sun [28], Mathews [74], Hu [75] and Purba [76], among others. They evaluated the performance of LLMs on vulnerability detection tasks, analyzed the gaps, and proposed ways to improve detection capabilities. Sun [77] et al. proposed GPTScan, which combines LLMs with static program analysis, while Li [78] et al. proposed LLift, which integrates LLMs with static analysis tools to improve the accuracy of detecting logic vulnerabilities. In addition, LLMs have been used for other security tasks, such as TitanFuzz [79] and FuzzGPT [80] by Deng et al. In addition to ChatAFL [81] for fuzzing and protocol testing. LLMs are also applied to program repair tasks such as ACFix [82] and ChatRepair.

VI. THREATS TO VALIDITY

Our study currently may have two limitations. The first involves the randomness of the LLM’s output. To reduce the bias produced by different parameters and prompts, we try to use default settings for models. We also analyze model performance using several runs and average the findings to assure the stability and reproducibility of the outcomes. The second constraint is the danger of data bias during the data gathering procedure, which may have an impact on the model evaluation outcomes. To address this issue, we employed a variety of data sources during the data gathering phase and rigorously screened and cleaned the data to reduce the influence of bias. In addition, we intend to include more control variables and experimental designs in future studies to help validate our findings. Nonetheless, these limitations remind us that we must exercise caution when interpreting our findings and conduct more extensive validation and cross-validation where possible to ensure the reliability and generalizability of our findings. We expect to address these constraints by continuously optimizing our methodology and tools in future works.

VII. CONCLUSION

In this research, we introduce SimilarGPT, a smart contract vulnerability detection tool that integrates Large Language Models with Code-based similarity checking. SimilarGPT effectively identify vulnerabilities in smart contracts by leveraging Ethereum’s prevalent code reuse issue. Our experimental results reveal that SimilarGPT excels at increasing the recall rate of vulnerability detection while decreasing the false positive rate. In particular, by

implementing the Socratic method, we significantly mitigating the false positives caused by LLMs hallucination. Furthermore, SimilarGPT allows it to adapt to the changing security ecosystem of smart contracts. Future work will focus on enhancing the code similarity identification method and expanding the dataset to increase the tool’s detection accuracy and usefulness.

REFERENCES

- [1] <https://ethereum.org/zh/>, accessed November 4, 2024.
- [2] <https://defillama.com/>, accessed November 4, 2024.
- [3] <https://defillama.com/hacks>, accessed November 4, 2024.
- [4] Z. Zhang, B. Zhang, W. Xu, and Z. Lin, “Demystifying exploitable bugs in smart contracts,” in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 615–627.
- [5] <https://github.com/Consensys/mythril>, accessed November 4, 2024.
- [6] <https://github.com/cryptic/slither>, accessed November 4, 2024.
- [7] C. F. Torres, A. K. Iannillo, A. Gervais, and R. State, “Confuzzius: A data dependency-aware hybrid fuzzer for smart contracts,” in *2021 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2021, pp. 103–119.
- [8] <https://github.com/enzyme-finance/oyente>, accessed November 4, 2024.
- [9] M. Rodler, W. Li, G. O. Karame, and L. Davi, “Sereum: Protecting existing smart contracts against re-entrancy attacks,” *arXiv preprint arXiv:1812.05934*, 2018.
- [10] B. Tan, B. Mariano, S. K. Lahiri, I. Dillig, and Y. Feng, “Soltype: refinement types for arithmetic overflow in solidity,” *Proceedings of the ACM on Programming Languages*, vol. 6, no. POPL, pp. 1–29, 2022.
- [11] S. Chaliasos, M. A. Charalambous, L. Zhou, R. Galanopoulou, A. Gervais, D. Mitropoulos, and B. Livshits, “Smart contract and defi security tools: Do they meet the needs of practitioners?” in *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, 2024, pp. 1–13.
- [12] J. Achiam, S. Adler, S. Agarwal, L. Ahmad, I. Akkaya, F. L. Aleman, D. Almeida, J. Altschmidt, S. Altman, S. Anadkat et al., “Gpt-4 technical report,” *arXiv preprint arXiv:2303.08774*, 2023.
- [13] W. X. Zhao, K. Zhou, J. Li, T. Tang, X. Wang, Y. Hou, Y. Min, B. Zhang, J. Zhang, Z. Dong et al., “A survey of large language models,” *arXiv preprint arXiv:2303.18223*, 2023.
- [14] Y. Sun, D. Wu, Y. Xue, H. Liu, H. Wang, Z. Xu, X. Xie, and Y. Liu, “Gptscan: Detecting logic vulnerabilities in smart contracts by combining gpt with program analysis,” in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 2024, pp. 1–13.
- [15] S. Hu, T. Huang, F. İlhan, S. F. Tekin, and L. Liu, “Large language model-powered smart contract vulnerability detection: New perspectives,” in *2023 5th IEEE International Conference on Trust, Privacy and Security in Intelligent Systems and Applications (TPS-ISA)*. IEEE, 2023, pp. 297–306.
- [16] Y. Sun, D. Wu, Y. Xue, H. Liu, W. Ma, L. Zhang, Y. Liu, and Y. Li, “Llm4vuln: A unified evaluation framework for decoupling and enhancing llms’ vulnerability reasoning,” *arXiv preprint arXiv:2401.16185*, 2024.
- [17] Y. Liu, Y. Xue, D. Wu, Y. Sun, Y. Li, M. Shi, and Y. Liu, “Propertygpt: Llm-driven formal verification of smart contracts through retrieval-augmented property generation,” *arXiv preprint arXiv:2405.02580*, 2024.
- [18] I. David, L. Zhou, K. Qin, D. Song, L. Cavallaro, and A. Gervais, “Do you still need a manual smart contract audit?” *arXiv preprint arXiv:2306.12338*, 2023.
- [19] F. Khan, I. David, D. Varro, and S. McIntosh, “Code cloning in smart contracts on the ethereum platform: An extended replication study,” *IEEE Transactions on Software Engineering*, vol. 49, no. 4, pp. 2006–2019, 2022.

- [20] M. Kondo, G. A. Oliva, Z. M. Jiang, A. E. Hassan, and O. Mizuno, "Code cloning in smart contracts: a case study on verified contracts from the ethereum blockchain platform," *Empirical Software Engineering*, vol. 25, pp. 4617–4675, 2020.
- [21] K. Sun, Z. Xu, C. Liu, K. Li, and Y. Liu, "Demystifying the composition and code reuse in solidity smart contracts," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2023, pp. 796–807.
- [22] J. Huang, S. Han, W. You, W. Shi, B. Liang, J. Wu, and Y. Wu, "Hunting vulnerable smart contracts via graph embedding based bytecode matching," *IEEE Transactions on Information Forensics and Security*, vol. 16, pp. 2144–2156, 2021.
- [23] H. Liu, Z. Yang, C. Liu, Y. Jiang, W. Zhao, and J. Sun, "Eclone: Detect semantic clones in ethereum via symbolic transaction sketch," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2018, pp. 900–903.
- [24] Z. Gao, L. Jiang, X. Xia, D. Lo, and J. Grundy, "Checking smart contracts with structural code embedding," *IEEE Transactions on Software Engineering*, vol. 47, no. 12, pp. 2874–2891, 2020.
- [25] F. Schär, "Decentralized finance: On blockchain-and smart contract-based financial markets," *FRB of St. Louis Review*, 2021.
- [26] L. Zhou, X. Xiong, J. Ernstberger, S. Chaliasos, Z. Wang, Y. Wang, K. Qin, R. Wattenhofer, D. Song, and A. Gervais, "Sok: Decentralized finance (defi) attacks," in *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2023, pp. 2444–2461.
- [27] T. Kojima, S. S. Gu, M. Reid, Y. Matsuo, and Y. Iwasawa, "Large language models are zero-shot reasoners," *Advances in neural information processing systems*, vol. 35, pp. 22 199–22 213, 2022.
- [28] Y. Sun, D. Wu, Y. Xue, H. Liu, W. Ma, L. Zhang, Y. Liu, and Y. Li, "Llm4vuln: A unified evaluation framework for decoupling and enhancing llms' vulnerability reasoning," *arXiv preprint arXiv:2401.16185*, 2024.
- [29] Z. Lin, M. Feng, C. N. d. Santos, M. Yu, B. Xiang, B. Zhou, and Y. Bengio, "A structured self-attentive sentence embedding," *arXiv preprint arXiv:1703.03130*, 2017.
- [30] P. Lewis, E. Perez, A. Piktus, F. Petroni, V. Karpukhin, N. Goyal, H. Küttler, M. Lewis, W.-t. Yih, T. Rocktäschel *et al.*, "Retrieval-augmented generation for knowledge-intensive nlp tasks," *Advances in Neural Information Processing Systems*, vol. 33, pp. 9459–9474, 2020.
- [31] Z. Ji, N. Lee, R. Frieske, T. Yu, D. Su, Y. Xu, E. Ishii, Y. J. Bang, A. Madotto, and P. Fung, "Survey of hallucination in natural language generation," *ACM Computing Surveys*, vol. 55, no. 12, pp. 1–38, 2023.
- [32] <https://immunefi.com/>, accessed November 4, 2024.
- [33] https://www.wikiwand.com/en/articles/Topological_sorting, accessed November 4, 2024.
- [34] <https://github.com/ConsensSys/surya>, accessed November 4, 2024.
- [35] <https://www.npmjs.com/package/@openzeppelin/contracts>, accessed November 4, 2024.
- [36] <https://huggingface.co/sentence-transformers/all-MiniLM-L6-v2>, accessed November 4, 2024.
- [37] https://www.sbert.net/docs/sentence_transformer/pretrained_models.html#scientific-similarity-models, accessed November 4, 2024.
- [38] https://www.sbert.net/docs/sentence_transformer/pretrained_models.html, accessed November 4, 2024.
- [39] <https://huggingface.co/sentence-transformers/all-mpnet-base-v2>, accessed November 4, 2024.
- [40] <https://huggingface.co/sentence-transformers/multi-qa-mpnet-base-dot-v1>, accessed November 4, 2024.
- [41] <https://huggingface.co/sentence-transformers/multi-qa-MiniLM-L6-cos-v1>, accessed November 4, 2024.
- [42] <https://github.com/SunWeb3Sec/DeFiHackLabs?tab=readme-ov-file>, accessed November 4, 2024.
- [43] <https://hacked.slowmist.io/en/>, accessed November 4, 2024.
- [44] <https://github.com/>, accessed November 4, 2024.
- [45] J. Qi, Z. Xu, Y. Shen, M. Liu, D. Jin, Q. Wang, and L. Huang, "The art of socratic questioning: Recursive thinking with large language models," *arXiv preprint arXiv:2305.14999*, 2023.
- [46] X. Sun, J. Li, Y. Zhong, D. Zhao, and R. Yan, "Towards detecting llms hallucination via markov chain-based multi-agent debate framework," *arXiv preprint arXiv:2406.03075*, 2024.
- [47] E. Y. Chang, "Prompting large language models with the socratic method," in *2023 IEEE 13th Annual Computing and Communication Workshop and Conference (CCWC)*. IEEE, 2023, pp. 0351–0360.
- [48] T. Chowdhury, C. Ling, X. Zhang, X. Zhao, G. Bai, J. Pei, H. Chen, and L. Zhao, "Knowledge-enhanced neural machine reasoning: A review," *arXiv preprint arXiv:2302.02093*, 2023.
- [49] <https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=smart+contract>, accessed November 4, 2024.
- [50] <https://solodit.xyz/>, accessed November 4, 2024.
- [51] <https://github.com/SunWeb3Sec/DeFiHackLabs/blob/main/past/2021/README.md#20210428-uranium---miscalculation>, accessed November 4, 2024.
- [52] <https://github.com/SunWeb3Sec/DeFiHackLabs/blob/main/past/2021/README.md#20210915-nimbus-platform>, accessed November 4, 2024.
- [53] <https://github.com/SunWeb3Sec/DeFiHackLabs/blob/main/past/2023/README.md#20230416-swapos-v2---error-k-value-attack>, accessed November 4, 2024.
- [54] L. Brent, N. Grech, S. Lagouvardos, B. Scholz, and Y. Smaragdakis, "Ethainter: a smart contract security analyzer for composite vulnerabilities," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2020, pp. 454–469.
- [55] L. Brent, A. Jurisevic, M. Kong, E. Liu, F. Gauthier, V. Gramoli, R. Holz, and B. Scholz, "Vandal: A scalable security analysis framework for smart contracts," *arXiv preprint arXiv:1809.03981*, 2018.
- [56] S. Kalra, S. Goel, M. Dhawan, and S. Sharma, "Zeus: analyzing safety of smart contracts," in *Ndss*, 2018, pp. 1–12.
- [57] T. D. Nguyen, L. H. Pham, J. Sun, Y. Lin, and Q. T. Minh, "sfuzz: An efficient adaptive fuzzer for solidity smart contracts," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 778–788.
- [58] B. Jiang, Y. Liu, and W. K. Chan, "Contractfuzzer: Fuzzing smart contracts for vulnerability detection," in *Proceedings of the 33rd ACM/IEEE international conference on automated software engineering*, 2018, pp. 259–269.
- [59] Y. Liu, Y. Li, S.-W. Lin, and C. Artho, "Finding permission bugs in smart contracts with role mining," in *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2022, pp. 716–727.
- [60] H. Wang, Y. Liu, Y. Li, S.-W. Lin, C. Artho, L. Ma, and Y. Liu, "Oracle-supported dynamic exploit generation for smart contracts," *IEEE Transactions on Dependable and Secure Computing*, vol. 19, no. 3, pp. 1795–1809, 2020.
- [61] J. Frank, C. Aschermann, and T. Holz, "{ETHBMC}: A bounded model checker for smart contracts," in *29th USENIX Security Symposium (USENIX Security 20)*, 2020, pp. 2757–2774.
- [62] A. Permenev, D. Dimitrov, P. Tsankov, D. Drachsler-Cohen, and M. Vechev, "Verx: Safety verification of smart contracts," in *2020 IEEE symposium on security and privacy (SP)*. IEEE, 2020, pp. 1661–1677.
- [63] S. So, M. Lee, J. Park, H. Lee, and H. Oh, "Verismart: A highly precise safety verifier for ethereum smart contracts," in *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2020, pp. 1678–1694.
- [64] H. Liu, C. Liu, W. Zhao, Y. Jiang, and J. Sun, "S-gram: towards semantic-aware security auditing for ethereum smart contracts," in *Proceedings of the 33rd ACM/IEEE international conference on automated software engineering*, 2018, pp. 814–819.
- [65] H. Liu, Z. Yang, Y. Jiang, W. Zhao, and J. Sun, "Enabling clone detection for ethereum via smart contract birthmarks," in *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*. IEEE, 2019, pp. 105–115.

- [66] G. A. Pierro and R. Tonelli, "Analysis of source code duplication in ethereum smart contracts," in *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2021, pp. 701–707.
- [67] X. Chen, P. Liao, Y. Zhang, Y. Huang, and Z. Zheng, "Understanding code reuse in smart contracts," in *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2021, pp. 470–479.
- [68] Z. Gao, L. Jiang, X. Xia, D. Lo, and J. Grundy, "Checking smart contracts with structural code embedding," *IEEE Transactions on Software Engineering*, vol. 47, no. 12, pp. 2874–2891, 2020.
- [69] S. Ullah, M. Han, S. Pujar, H. Pearce, A. Coskun, and G. Stringhini, "Llms cannot reliably identify and reason about security vulnerabilities (yet?): A comprehensive evaluation, framework, and benchmarks," *arXiv preprint arXiv:2312.12575*, 2023.
- [70] M. Fu, C. K. Tantithamthavorn, V. Nguyen, and T. Le, "Chatgpt for vulnerability detection, classification, and repair: How far are we?" in *2023 30th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 2023, pp. 632–636.
- [71] C. Thapa, S. I. Jang, M. E. Ahmed, S. Camtepe, J. Pieprzyk, and S. Nepal, "Transformer-based language models for software vulnerability detection," in *Proceedings of the 38th Annual Computer Security Applications Conference*, 2022, pp. 481–496.
- [72] I. David, L. Zhou, K. Qin, D. Song, L. Cavallaro, and A. Gervais, "Do you still need a manual smart contract audit?" *arXiv preprint arXiv:2306.12338*, 2023.
- [73] M. Alqarni and A. Azim, "Low level source code vulnerability detection using advanced bert language model." in *Canadian AI*, 2022.
- [74] N. S. Mathews, Y. Brus, Y. Aafer, M. Nagappan, and S. McIntosh, "Llbezpeky: Leveraging large language models for vulnerability detection," *arXiv preprint arXiv:2401.01269*, 2024.
- [75] S. Hu, T. Huang, F. İlhan, S. F. Tekin, and L. Liu, "Large language model-powered smart contract vulnerability detection: New perspectives," in *2023 5th IEEE International Conference on Trust, Privacy and Security in Intelligent Systems and Applications (TPS-ISA)*. IEEE, 2023, pp. 297–306.
- [76] M. D. Purba, A. Ghosh, B. J. Radford, and B. Chu, "Software vulnerability detection using large language models," in *2023 IEEE 34th International Symposium on Software Reliability Engineering Workshops (ISSREW)*. IEEE, 2023, pp. 112–119.
- [77] Y. Sun, D. Wu, Y. Xue, H. Liu, H. Wang, Z. Xu, X. Xie, and Y. Liu, "Gptscan: Detecting logic vulnerabilities in smart contracts by combining gpt with program analysis," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 2024, pp. 1–13.
- [78] H. Li, Y. Hao, Y. Zhai, and Z. Qian, "The hitchhiker's guide to program analysis: A journey with large language models," *arXiv preprint arXiv:2308.00245*, 2023.
- [79] Y. Deng, C. S. Xia, H. Peng, C. Yang, and L. Zhang, "Large language models are zero-shot fuzzers: Fuzzing deep-learning libraries via large language models," in *Proceedings of the 32nd ACM SIGSOFT international symposium on software testing and analysis*, 2023, pp. 423–435.
- [80] Y. Deng, C. S. Xia, C. Yang, S. D. Zhang, S. Yang, and L. Zhang, "Large language models are edge-case generators: Crafting unusual programs for fuzzing deep learning libraries," in *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, 2024, pp. 1–13.
- [81] R. Meng, M. Mirchev, M. Böhme, and A. Roychoudhury, "Large language model guided protocol fuzzing," in *Proceedings of the 31st Annual Network and Distributed System Security Symposium (NDSS)*, 2024.
- [82] L. Zhang, K. Li, K. Sun, D. Wu, Y. Liu, H. Tian, and Y. Liu, "Acfix: Guiding llms with mined common rbac practices for context-aware repair of access control vulnerabilities in smart contracts," *arXiv preprint arXiv:2403.06838*, 2024.