GPTScan: Detecting Logic Vulnerabilities in Smart Contracts by Combining GPT with Program Analysis

Yuqiang Sun
Nanyang Technological University
Singapore, Singapore
suny0056@e.ntu.edu.sg

Daoyuan Wu*
Nanyang Technological University
Singapore, Singapore
daoyuan.wu@ntu.edu.sg

Yue Xue
MetaTrust Labs
Singapore, Singapore
xueyue@metatrust.io

Han Liu
East China Normal University
Shanghai, China
hanliu@stu.ecnu.edu.cn

Haijun Wang
Xi'an Jiaotong University
Xi'an, China
haijunwang@xjtu.edu.cn

Zhengzi Xu
Nanyang Technological University
Singapore, Singapore
zhengzi.xu@ntu.edu.sg

Xiaofei Xie
Singapore Management University
Singapore, Singapore
xfxie@smu.edu.sg

Yang Liu
Nanyang Technological University
Singapore, Singapore
yangliu@ntu.edu.sg

ABSTRACT

Smart contracts are prone to various vulnerabilities, leading to substantial financial losses over time. Current analysis tools mainly target vulnerabilities with fixed control- or data-flow patterns, such as re-entrancy and integer overflow. However, a recent study on Web3 security bugs revealed that about 80% of these bugs cannot be audited by existing tools due to the lack of domain-specific property description and checking. Given recent advances in Large Language Models (LLMs), it is worth exploring how Generative Pre-training Transformer (GPT) could aid in detecting *logic vulnerabilities*.

In this paper, we propose GPTScan, the first tool combining GPT with static analysis for smart contract logic vulnerability detection. Instead of relying solely on GPT to identify vulnerabilities, which can lead to high false positives and is limited by GPT's pre-trained knowledge, we utilize GPT as a versatile code understanding tool. By breaking down each logic vulnerability type into *scenarios* and *properties*, GPTScan matches candidate vulnerabilities with GPT. To enhance accuracy, GPTScan further instructs GPT to intelligently recognize key variables and statements, which are then validated by static confirmation. Evaluation on diverse datasets with around 400 contract projects and 3K Solidity files shows that GPTScan achieves high precision (over 90%) for token contracts and acceptable precision (57.14%) for large projects like Web3Bugs. It effectively detects ground-truth logic vulnerabilities with a recall of over 70%, including 9 new vulnerabilities missed by human auditors. GPTScan is fast and cost-effective, taking an average of 14.39 seconds and 0.01 USD to scan per thousand lines of Solidity code. Moreover, static confirmation helps GPTScan reduce two-thirds of false positives.

1 INTRODUCTION

Smart contracts have emerged as the cornerstone of decentralized finance (DeFi), providing a programmable and automated solution for executing financial transactions. However, the security of these smart contracts has become a major concern due to various security breaches [1, 4]. These breaches have led to financial losses amounting to billions of dollars [66]. This situation is a disaster for DeFi service providers, posing a significant threat to the entire DeFi ecosystem and the safety of users' assets.

Despite the availability of numerous analysis tools [29, 30, 37, 43, 56], they often focus on vulnerabilities with fixed control- or data-flow patterns, such as re-entrancy [52, 61], integer overflow [54], and access control vulnerabilities [36, 39, 46]. However, a recent study conducted by Zhang et al. [65] on Web3 security bugs reveals that around 80% of these vulnerabilities remain undetected by existing tools. These undetected vulnerabilities are primarily associated with the business logic of smart contracts. Traditional static and dynamic analysis schemes, such as Slither [37], do not effectively address these vulnerabilities in smart contracts because they do not aim to comprehend the underlying business logic of smart contracts, nor do they model the functionality or consider the roles of various variables or functions.

In this paper, we explore how recent advances in Large Language Models (LLMs) [5] or Generative Pre-training Transformer (GPT) [44, 49] could aid in detecting logic vulnerabilities in smart contracts. A recent technical report [34] attempted to use GPT by providing it with high-level vulnerability descriptions for project-wide "Yes-or-No" inquiries, which is already easier than typical function-level vulnerability detection. However, this approach suffered from a high false positive rate of around 96% and required advanced reasoning capabilities from GPT, necessitating the use of GPT-4 instead of GPT-3.5. Instead, we treat GPT as a generic and powerful code understanding tool and investigate how this capability can be combined with static analysis to create an intelligent detection system for logic vulnerabilities.

To this end, we propose GPTScan, the first tool that combines GPT with static analysis for detecting logic vulnerabilities in smart contracts. To leverage GPT's code understanding capability, we break down each logic vulnerability type into code-level scenarios and properties. *Scenarios* describe the code functionality under which a logic vulnerability could occur, while *properties* explain the vulnerable code attributes or operations. This approach enables GPTScan to directly match candidate vulnerable functions based

*Corresponding author.

on code-level semantics. However, since GPT-based matching is still coarse-grained, GPTScan further instructs GPT to intelligently recognize key variables and statements, which are then validated by dedicated static confirmation modules. Moreover, a smart contract project can consist of multiple Solidity files, making it infeasible or costly to directly feed all of them to GPT. To address this issue, GPTScan employs a multi-dimensional filtering process to effectively narrow down the candidate functions for GPT matching.

We implemented GPTScan with the widely used GPT-3.5-turbo model [27], which is 20 times more cost-effective [6] than the advanced GPT-4 model. Moreover, our multi-dimensional filtering allowed GPTScan to utilize the default 4k context token size instead of 16k, resulting in a more economical solution. The parameters were mainly kept at their default values, except for the `temperature` parameter, which was adjusted from the default value of 1 to 0 to reduce the impact of GPT's output randomness. To further enhance the reliability of GPT's answers and minimize the influence of output randomness, we proposed a trick called "mimic-in-the-background" prompting, inspired by the success of zero-shot chain-of-thought prompting [44]. For the static analysis part, GPTScan relies on ANTLR [21] and crytic-compiler [7] to support call graph and data dependency analysis.

To comprehensively evaluate GPTScan under different scenarios, we collected three diverse datasets from real-world smart contracts. Together, these datasets comprise around 400 contract projects, 3K Solidity files, 472K lines of code, and include 62 ground-truth logic vulnerabilities. The first dataset, named *Top200*, consists of smart contracts with the top 200 market capitalization. This dataset primarily serves to evaluate the false positive rate of GPTScan. The second dataset, referred to as *Web3Bugs*, was collected from the recent Web3Bugs dataset [8]. The third dataset, called *DefiHacks*, is sourced from the well-known DeFi Hacks dataset [9], which contains vulnerable contracts that have experienced past attack incidents. *Top200* and *DefiHacks* primarily comprise cryptocurrency token contract projects, whereas *Web3Bugs* consists of large contract projects audited on the Code4rena platform [10], with an average of 36 Solidity files per project.

GPTScan achieves a low false positive rate of 4.39% when analyzing non-vulnerable top contracts like *Top200*. It also demonstrates similar performance in analyzing another set of token contracts, *DefiHacks*, with a precision of 90.91%. These results indicate that GPTScan is suitable for massive scanning of on-chain contracts. Moreover, when analyzing large contract projects in *Web3Bugs*, GPTScan still achieves an acceptable precision of 57.14%. Furthermore, GPTScan shows its efficacy in detecting ground-truth logic vulnerabilities in the *Web3Bugs* and *DefiHacks* datasets, with a recall of 83.33% and an F1 score of 67.8% for *Web3Bugs*, and a recall of 71.43% and an F1 score of 80% for *DefiHacks*. In particular, GPTScan identifies 9 new vulnerabilities that were not present in the audit reports of Code4rena. This highlights the value of GPTScan as a useful supplement to human auditors.

A further analysis of GPTScan's running logs reveals that GPTScan is fast and cost-effective, taking an average of only 14.39 seconds and 0.01 USD to scan per thousand lines of Solidity code in the tested datasets. The relatively higher cost (around 0.018 USD) and slower speed (around 20 seconds) observed for *Web3Bugs* and *DefiHacks* can be attributed to the presence of more complex functions that cannot be filtered out by static filtering and scenario matching. Furthermore, we diagnose that GPTScan's static confirmation reduces 65.84% of the original false positive cases in the *Web3Bugs* dataset. This finding underscores the importance of combining GPT with static analysis to achieve accurate results.

Availability. GPTScan has been integrated as a part of MetaScan (https://metatrust.io/metascan), an industry-leading smart contract security scanning platform [22, 25]. Moreover, GPTScan's evaluation data is available at https://sites.google.com/view/gptscan for facilitating easier comparisons in future work.

Roadmap. The rest of this paper is organized as follows. In §2, we introduce some background information. In §3, we motivate the need of both GPT and static analysis. Following that, in §4, we detail the design of GPTScan, while in §5, we evaluate its performance. We then discuss the applicability and current limitations in §6. Finally, we summarize related work in §7 and conclude in §8.

2 BACKGROUND

In this section, we introduce some background about smart contract vulnerabilities and GPT's application in vulnerability detection.

Smart contract vulnerability types. Smart contracts are self-running programs deployed on blockchain, written in a high-level language called Solidity [11]. As described by Zhang et al. [65], there are 26 types of vulnerabilities in smart contracts, categorized into 3 groups. The vulnerabilities in the first group are hard to exploit, doubtful, or not directly related to the functionalities of a given project. The second group of vulnerabilities involves the use of simple and general oracles, not requiring an in-depth understanding of the code semantics. Examples include *Re-entrancy* and *Arithmetic Overflow*. Such vulnerabilities can be detected by data flow tracing (e.g., Slither [37]), static symbolic execution (e.g., Solidity SMT Checker [12] and Mythril [13]) and other static analysis tools [29, 43, 47]. The third group of vulnerabilities requires high-level semantical oracles for detection and is closely related to the business logic. Most of these vulnerabilities are not detectable by existing static analysis tools. This group comprises six main types of vulnerabilities: (S1) price manipulation, (S2) ID-related violations, (S3) erroneous state updates, (S4) atomicity violation, (S5) privilege escalation, and (S6) erroneous accounting.

GPT and its application in vulnerability detection. Generative Pre-training Transformer (GPT) models, such as GPT-3.5 [49], are large language models (LLMs) trained on vast text corpora, including source code descriptions of different programming languages and vulnerabilities. With this knowledge, GPT can understand and interpret source code, enabling zero-shot learning [44], where examples of vulnerabilities are not needed to detect vulnerabilities in source code. However, GPT still has a long way to go before it can fully replace humans in code auditing [14]. David et al. [34] provided GPT with vulnerability descriptions and used them to detect vulnerabilities in source code. They fed the entire project into the GPT-4-32k model to detect 38 types of vulnerabilities in smart contracts. However, the results were unsatisfactory and even worse than a random model in terms of recall. Due to the limitations of the GPT model on content length (from 4k tokens in GPT-3.5 to 32k tokens in GPT-4), analyzing complete projects or documents using GPT is not viable, making David et al.'s approach unsuitable

```
1  function deposit(uint256 _amount) external returns (uint256) {
2      uint256 _pool = balance();
3      uint256 _before = token.balanceOf(address(this));
4      token.safeTransferFrom(msg.sender, address(this), _amount);
5      uint256 _after = token.balanceOf(address(this));
6      _amount = _after.sub(_before); // Additional check for
           deflationary tokens
7      uint256 _shares = 0;
8      if (totalSupply() == 0) {
9          _shares = _amount;
10     } else {
11         _shares = (_amount.mul(totalSupply())).div(_pool);
12     }
13     _mint(msg.sender, _shares);
14 }
```

Figure 1: The *Risky First Deposit* (line 8-9) vulnerability.

```
1  function transfer(address account, uint256 amount) external
       override notPaused returns (bool) {
2      require(msg.sender != account, Error.
           SELF_TRANSFER_NOT_ALLOWED);
3      require(balances[msg.sender] >= amount, Error.
           INSUFFICIENT_BALANCE);
4      // Initialize the ILiquidityPool pool variable
5      pool.handleLpTokenTransfer(msg.sender, account, amount);
6      balances[msg.sender] -= amount;
7      balances[account] += amount;
8      address lpGauge = currentAddresses[_LP_GAUGE];
9      if (lpGauge != address(0)) {
10         ILpGauge(lpGauge).userCheckpoint(msg.sender);
11         ILpGauge(lpGauge).userCheckpoint(account);
12     }
13     emit Transfer(msg.sender, account, amount);
14     return true;
15 }
```

Figure 2: The *Wrong Checkpoint Order* (line 6-7 & line 10-11).

for large projects. Moreover, as GPT has limited logical reasoning capabilities, its results may not always be accurate, necessitating verification using other methods to reduce the false positive rate.

3 MOTIVATING EXAMPLES

In this section, we use two real-world smart contract examples to motivate why both GPT and static analysis are needed in the process of detecting logic vulnerabilities.

Example 1: Requiring GPT to recognize variables and static analysis to confirm the variable dependency. The first example in Figure 1 is from the Code4rena [10] project *2021-11-yaxis* [2]. The vulnerability occurs when the LP (Liquidity Pool [45]) token's entire share is minted to the first depositor (line 9) while the current LP token supply is zero (line 8). Consequently, the first depositor can arbitrarily inflate the price per LP share (e.g., from a small `_amount` to an extremely large value; see the detail of an exploit in GitHub issue[15]), leading to future token deposits from victim users to be indirectly "occupied" by the first depositor. While static analysis may use hard-coded patterns to detect the `totalSupply()` logic in line 8, GPT is necessary to intelligently recognize the variables responsible for holding the deposit amount (`_amount`) and the total share of the pool (`_shares`) to avoid false positives. Nevertheless, precisely validating the vulnerable logic from line 8 to 9 falls outside the scope of GPT, making static analysis essential for this task.

Example 2: Requiring GPT to recognize statements and static analysis to confirm the statement order. The second example in Figure 2 is from the Code4Rena project *2022-04-backd* [16], where the executing order of some statements is incorrect. The correct order should be to first perform user checkpoints (line 10-11) and then update the balances of the sender and receiver for the transfer (lines 6-7). Due to this mistake, a user can steal all rewards because the checkpoint is executed after reward transfer [17]. To detect this vulnerability, GPT is required to understand the semantic of statements and recognize those that perform user checkpoints and those that change user balances. However, we found that GPT struggles to comprehend the concept of "before," and as a result, relying solely on GPT could report a patched version [18] of the `transfer` function as vulnerable. Static analysis is thus necessary.

Based on the above examples, we find that static analysis cannot understand high-level semantic information, and GPT may overlook some low-level information, potentially leading to low recall and high false positives, respectively. Combining these two techniques can complement each other and enhance detection performance.

4 GPTSCAN

In this section, we present GPTScan's overall design and its three core components from §4.1 to §4.4, followed by a summary of some key implementation details in §4.5.

4.1 Overview and Challenges

Figure 3 illustrates GPTScan's high-level workflow, with blue blocks denoting GPT tasks and green blocks representing static analysis. Given a smart contract project, which could be a standalone Solidity file or a framework-based contract project containing multiple Solidity files, GPTScan first performs contract parsing, call graph analysis to determine function reachability, and comprehensive filtering to extract candidate functions and their corresponding context functions. GPTScan then utilizes GPT to match the candidate functions with pre-abstracted scenarios and properties of relevant vulnerability types. For the matched functions, GPTScan further recognizes their key variables and statements via GPT, which are subsequently passed to specialized static analysis modules for vulnerability confirmation.

During this three-step process, we need to address the following three challenges:

C1: A smart contract project may contain tens of Solidity files[1], making it infeasible or costly to directly feed all of them to GPT. Moreover, the presence of non-vulnerable functions may affect GPT's recognition of vulnerable ones. Therefore, *how to effectively narrow down the candidate functions for GPT matching becomes essential.*

C2: Existing GPT-based vulnerability detection works [14, 34, 35] typically feed GPT with high-level vulnerability descriptions for vulnerability matching, which either demands advanced reasoning capabilities from GPT or relies on the pre-trained vulnerability knowledge of GPT models. Hence, *can we break down vulnerability types in a manner that allows GPT, as a generic and intelligent code understanding tool, to recognize them directly from code-level semantics?*

[1] According to our evaluation in §5, a Code4rena project has 36 Solidity files on average. In contrast to a recent study [34], which claimed to feed entire contracts to the GPT-4 model with 32k tokens, we cannot feed the entire project into the model for analysis.

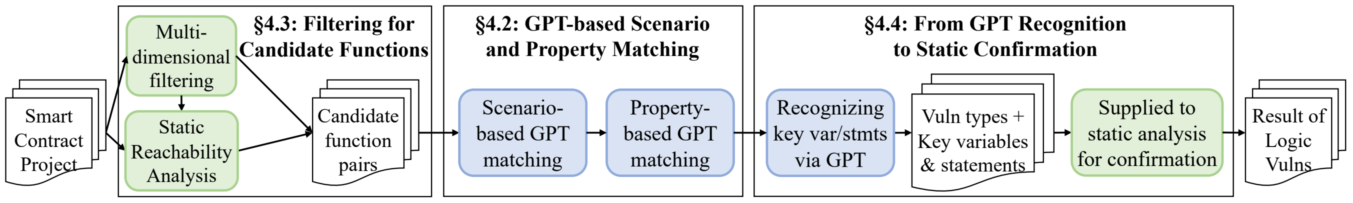
Smart Contract Project
Multi-dimensional filtering
Static Reachability Analysis
§4.3: Filtering for Candidate Functions
Candidate function pairs
§4.2: GPT-based Scenario and Property Matching
Scenario-based GPT matching
Property-based GPT matching
§4.4: From GPT Recognition to Static Confirmation
Recognizing key var/stmts via GPT
Vuln types + Key variables & statements
Supplied to static analysis for confirmation
Result of Logic Vulns

Figure 3: A high-level overview of GPTScan, with blue blocks denoting GPT tasks and green blocks representing static analysis.

Table 1: Breaking down ten common logic vulnerability types into scenarios and properties.

Vulnerability Type	Scenario and Property	Filtering Type	Static Check
Approval Not Cleared	**Scenario:** add or check approval via require/if statements before the token transfer **Property:** and there is no clear/reset of the approval when the transfer finishes its main branch or encounters exceptions	FNI, FCCE	VC
Risky First Deposit	**Scenario:** deposit/mint/add the liquidity pool/amount/share **Property:** and set the total share to the number of first deposit when the supply/liquidity is 0	FCCE	DF, VC
Price Manipulation by AMM	**Scenario:** have code statements that get or calculate LP token's value/price **Property:** based on the market reserves/AMMprice/exchangeRate OR the custom token balanceOf/totalSupply/amount/liquidity calculation	FNK, FCCE	DF
Price Manipulation by Buying Tokens	**Scenario:** buy some tokens **Property:** using Uniswap/PancakeSwap APIs	FNK, FCE	FA
Vote Manipulation by Flashloan	**Scenario:** calculate vote amount/number **Property:** and this vote amount/number is from a vote weight that might be manipulated by flashloan	FCCE	DF
Front Running	**Scenario:** mint or vest or collect token/liquidity/earning and assign them to the address recipient or to variable **Property:** and this operation could be front run to benefit the account/address that can be controlled by the parameter and has no sender check in the function code	FNK, FPNC, FPT, FCNE, FNM	FA
Wrong Interest Rate Order	**Scenario:** have inside code statements that update/accrue interest/exchange rate **Property:** and have inside code statements that calculate/assign/distribute the balance/share/stake/fee/loan/reward	FCE, CEN	OC
Wrong Checkpoint Order	**Scenario:** have inside code statements that invoke user checkpoint **Property:** and have inside code statements that calculate/assign/distribute the balance/share/stake/fee/loan/reward	FCE, CEN	OC
Slippage	**Scenario:** involve calculating swap/liquidity or adding liquidity, and there is asset exchanges or price queries **Property:** but this operation could be attacked by Slippage/Sandwich Attack due to no slip limit/minimum value check	FCCE, FCNCE	VC
Unauthorized Transfer	**Scenario:** involve transfering token from an address different from message sender **Property:** and there is no check of allowance/approval from the address owner	FNK, FCNE, FCE, FCNCE, FPNC	VC

C3: Considering that GPT may produce unreliable answers or fail to recognize differences in similar functions, *further confirming the matched potential vulnerabilities becomes critical.*

Since challenge C1 and C3 are both related to challenge C2, we first present how we tackle C2 in §4.2, followed by our solutions to C1 and C3 in §4.3 and §4.4, respectively.

4.2 GPT-based Scenario and Property Matching

Existing GPT-based vulnerability detection works [14, 34, 35] identify vulnerabilities by simply feeding GPT with high-level vulnerability descriptions, such as the one provided for the Front Running vulnerability: "*An attack where an attacker observes pending transactions and creates a new transaction with a higher gas price, enabling it to be processed before the observed transaction. This is often done to gain an unfair advantage in decentralized exchanges or other time-sensitive operations.*" [34]. However, these descriptions are condensed from root causes rather than code properties, making it challenging for GPT to directly interpret code-level semantics.

Breaking down vulnerabilities into scenarios and properties. GPTScan adopts a different approach by breaking down vulnerability types into code-level scenarios and properties. Specifically, we use *scenarios* to describe the code functionality under which a logic vulnerability could occur and *properties* to explain the vulnerable code attributes or operations. Table 1 showcases how we break down ten common logic vulnerability types into scenarios and properties. These vulnerability types were selected from a recent study [65] on smart contract vulnerabilities that require high-level semantic oracles [8]. The study summarizes six categories of logic vulnerabilities from S1 to S6 (see §2), and we chose ten representative cases from these categories. For instance, the Approval Not Cleared vulnerability is from S3, which involves missing state update, and the two wrong order vulnerabilities are from S6, relating to incorrect calculating order. Note that in this paper, we manually broke down ten vulnerability types to precisely describe their code-level attributes. To support more logic vulnerability types in future work, we have figured out a GPT-based approach. This approach employs GPT-4 to automatically extract initial scenario and property sentences from past vulnerability reports, validate them using the original vulnerable code, and iteratively regenerate new sentences until a scenario and property sentence pass the original vulnerability validation. However, while the generation

Prompt Template

System: You are a smart contract auditor. You will be asked questions related to code properties. You can mimic answering them in the background five times and provide me with the most frequently appearing answer. Furthermore, please strictly adhere to the output format specified in the question; there is no need to explain your answer.

Scenario Matching
Given the following smart contract code, answer the questions below and organize the result in a json format like {"1": "Yes" or "No", "2": "Yes" or "No"}.
"1": **[%SCENARIO_1%]**?
"2": **[%SCENARIO_2%]**?
[%CODE%]

Property Matching
Does the following smart contract code "**[%SCENARIO, PROPERTY%]**"? Answer only "Yes" or "No".
[%CODE%]

Figure 4: Prompt for scenario and property matching.

of scenario and property sentences can be automated, the prompt used for GPT recognition, which we will explain in §4.4, must be manually crafted for different types of vulnerabilities.

Each scenario and property can be divided into two parts. The first part includes a description of the function's functionality, which helps GPTScan perform an initial screening of candidate functions to reduce unnecessary subsequent scanning. Using *Front Running* as an example, functions affected by this vulnerability type must involve actions like minting, vesting, or transferring tokens of other users. The approval for such actions is granted in a previous transaction, allowing attackers to front-run the function and gain an unfair advantage. The second part includes a description of the function's behavior, which is related to the root cause of the vulnerabilities, such as the lack of security checks and incorrect accounting order. If a function meets the properties of the first part, i.e., scenarios, GPTScan will send the function to GPT again to check if it satisfies both the scenarios and properties. If both parts are satisfied, GPTScan considers the function likely to contain a specific type of vulnerability and will confirm it in the later steps.

Yes-or-No scenario and property matching. With the abstracted scenarios and properties, we utilize them to match candidate functions using GPT. Figure 4 shows the prompt template employed by GPTScan for scenario and property matching, which is designed with three considerations. Firstly, property matching is performed only for functions that pass our scenario matching. This separation of scenario and property enables us to query all scenarios in a single prompt, thus saving on GPT costs. Secondly, during property matching, we double-confirm the scenario with GPT by querying the combination of scenario and property rather than property alone. Indeed, the scenarios and properties from Table 1 are designed to form a complete sentence. Thirdly, considering that GPT models sometimes provide ambiguous answers or hard-to-parse text, scenario and property matching are designed with yes or no questions only, aiming to minimize the impact of unstructured GPT responses. Moreover, we instruct GPT to learn the output JSON format for the multiple-choice scenario matching, leveraging GPT's instruction learning capability [50].

Minimizing the impact of GPT output randomness. Although we use yes-or-no questions to restrict the format of GPT responses, it does not eliminate the inherent randomness of GPT model output. Consequently, GPT may not provide the same answer for the same question. To address this, one approach is to set the `temperature` parameter of GPT models to 0, making the model tend to be deterministic. To further enhance the reliability of the answer and minimize the influence of GPT output randomness, we propose a trick called "mimic-in-the-background" prompting, which is inspired by the successful usage of "*Let's think step by step.*" in the zero-shot chain-of-thought prompting [44] – evaluating such prompting is beyond the scope of this paper. As shown in Figure 4, we use a GPT system prompt to instruct the model to mimic answering questions in the background five times and provide the most frequently appearing answer to ensure greater consistency.

4.3 Multi-dimensional Function Filtering

As mentioned in §4.1, we need to filter the candidate functions before GPT matching. Here, we propose a multi-dimensional filtering to systematically select candidate functions for different vulnerability types. Moreover, we conduct reachability analysis to retain only the functions that could be accessed by potential attackers.

Project-wide file filtering. Our multi-dimensional filtering begins with project-wide file filtering, which involves excluding non-Solidity files e..g, those under the "node_modules" directory, test files (e.g., those found in various "test" directories), and third-party library files (e.g., those from well-known libraries such as "openzeppelin", "uniswap", and "pancakeswap"). Once these files are filtered out, GPTScan can concentrate on the project's Solidity files themselves.

Filtering out OpenZeppelin functions. OpenZeppelin [26] provides a set of libraries to build secure smart contracts on Ethereum, widely used in the smart contract community. While we have filtered out OpenZeppelin contracts imported as libraries, we found that OpenZeppelin functions are often directly copied into many developers' contract code, making our project-wide file filtering ineffective. To address this, we first perform an offline analysis of OpenZeppelin's source code to extract all its API function signatures as a whitelist. Each function signature in the whitelist includes the access control modifier, the class name (sub-contract name), function name, return value types, and parameter types. For example, the signature of the `transfer` function in the `ERC20` contract is `public ERC20.transfer(address,uint256)`. Next, GPTScan generates the signature of all candidate functions in the same format and compares them with the signatures in the whitelist. Note that the signature of the candidate function is generated with both the class name and the name of the inherited class because developers may implement the inherited class. By conducting this comparison, GPTScan excludes functions with the same signature as those in the whitelist, which we consider secure in this paper. In the future, we will add clone-based filtering that covers function bodies.

Vulnerability-specific function filtering. After project-wide file and OpenZeppelin filtering, GPTScan conducts function-level filtering for different vulnerability types, which constitutes the major part of GPTScan's multi-dimensional filtering. To accommodate various filtering requirements, we have designed a YAML-based [3] filtering rule specification to support the following filtering rules:
FNK: The Function Name should contain at least one Keyword.
FCE: The Function Content should contain at least one Expression.
FCNE: The Function Content should Not contain any Expression.
FCCE: The Function Content should contain at least one Combination of given Expressions.
FCNCE: The Function Content should Not contain any Combination of given Expressions.
FPT: The Function Parameters should match the given Types.
FPNC: The Function should be Public, and we will Not analyze it with its Caller.
FNM: The Function should Not contain Modifiers that with access control (e.g., `onlyOwner`).
CFN: The Callers of this Function will Not be analyzed.

These filtering rules encompass the basic function name (FNK), the detailed function content (FCE, FCNE, FCCE, and FCNCE), the function parameters (FPT), and the function's caller relation (FPNC, FNM, CFN). Different vulnerabilities will utilize their specific filtering rules. The selection of filters is mainly based on the domain knowledge of the vulnerability types. For example, the Risky First Deposit vulnerability shown in Figure 1 uses only the FCCE rule type to select any combination of "total," "supply," and "liquidity," either separately or together, to ensure that the deposit is related to the calculation of total supply or liquidity of the token. On the other hand, *Price Manipulation by AMM* is related to the calculation of token prices. In this rule, we used the FNK rule to select functions related to price calculation, and the FCE rule to select functions that contain the keywords "price," "value," and "liquidity."

Reachability analysis. After filtering, we perform call graph analysis to determine the reachability of candidate functions. We utilize ANTLR [21], a lexer and parser generator, to parse the source code of the smart contract project and generate an abstract syntax tree (AST). Using the AST, we build a call graph for the entire project. In Solidity, there are four types of access control annotations: `public`, `external`, `internal` and `private`. Functions marked as `public` and `external` can be called by anyone, making them directly reachable for potential attackers. Functions marked as `internal` and `private` might be called by other reachable functions, so we analyze their reachability and include them if they are reachable. Moreover, Solidity allows developers to use custom modifiers to perform permission checks before function calls. For example, functions annotated with `onlyOwner` are only allowed to be called by the owner, which we consider as unreachable. Functions that are deemed unreachable are excluded from the subsequent GPT-based matching in §4.2.

An Example Prompt for GPT Recognition

System: (same as in Figure 4, omitted here for brevity.)

In this function, which variable or function holds the total supply/liquidity AND is used by the conditional branch to determine the supply/liquidity is 0? Please answer in a section starts with "VariableB:".
In this function, which variable or function holds the value of the deposit/mint/add amount? Please answer in a section starts with "VariableC:".
Please answer in the following json format: {"VariableA":{"Variable name":"Description"}, "VariableB":{"Variable name":"Description"}, "VariableC":{"Variable name":"Description"}}
[%CODE%]

Figure 5: A prompt for finding related variables/statements.

4.4 From GPT Recognition to Static Confirmation

Although the candidate functions pass the initial filtering and GPT matching on function properties, GPT does not always pay attention to syntactic details, such as conditional statements, require statements, assert statements, revert statements, etc. A more fine-grained static analysis is necessary to identify potentially vulnerable functions at this stage. Static analysis tools typically focus on specific variables or statements, while our current inputs are still functions. This is where we need the assistance of GPT to extract the variables and statements related to the specific business logic described in the prompt. With these variables and statements, we can use static analysis to confirm whether the vulnerability exists or not. An example of the prompt sent to GPT to ask for related variables or expressions for *Risky First Deposit* is shown in Figure 5.

For each extracted variable or statement, GPTScan instructs GPT to provide a short description. This description helps determine whether the given variables are relevant to the problem and helps avoid incorrect answers. If GPT provides variables or statements that do not exist in the context of the function or if the description is not relevant to the question asked, GPTScan terminates the judgment process and considers that the vulnerability does not exist. On the other hand, if the provided variables and statements pass validation, GPTScan feeds them into a static analysis tool to confirm the existence of the vulnerability using methods such as static data flow tracing and static symbolic execution. Specifically, we have designed the following four major types of static analysis to validate the common logic vulnerabilities listed in Table 1:
Static Data Flow Tracing (DF): This method traces the data flow of variables in the program, where static analysis determines whether the two variables or expressions provided by GPT have data dependencies. For example, Figure 1 shows that data flow analysis is needed to determine whether the share is calculated directly with the deposit amount in the *Risky First Deposit* vulnerability.
Value Comparison Check (VC): This method checks whether two variables or expressions are compared in condition statements,

such as *require*, *assert*, and *if*. It is used to ensure that variables or expressions are properly checked before usage. In *Risky First Deposit*, VC is used to check whether the share is compared with the deposit amount. Likewise, in *Unauthorized Transfer*, VC is used to verify whether the sender has been checked before the transfer.
Order Check (OC): This method checks the execution order of two statements, where static analysis determines the order of two statements provided by GPT. For example, Figure 2 shows that OC is used to verify the execution order of performing a transfer and updating the checkpoint in *Wrong Checkpoint Order*.
Function Call Argument Check (FA): This method checks whether an argument of a function call can be controlled by the user or meets specific requirements. Specifically, GPT provides a function call and the index of an argument, and static analysis determines whether the argument can be controlled by the user or meets the requirements described in the rules. In *Price Manipulation by Buying Tokens*, the function calls need to be checked with FA, as some sensitive variables may be used as parameters and cause price manipulation.

4.5 Implementation

GPTScan is implemented with 3,640 lines of code (LOC) in Python and 154 LOC in Java/Kotlin. In this section, we provide a summary of some key implementation details as follows.

GPT model and its parameters. During the development and testing of GPTScan, we utilized OpenAI's GPT-3.5-turbo model [27]. Thanks to the multi-dimensional filtering introduced in §4.3, GPTScan could use the default 4k context token size instead of 16k, which resulted in a more cost-effective solution. The parameters were mainly kept at their default values, including `TopP` set to 1, Frequency Penalty set to 0, and Presence Penalty set to 0. As discussed in §4.2, we adjusted the `temperature` parameter from the default value of 1 to 0 to minimize the impact of GPT output randomness. During each GPT query, the question is sent with an empty session to ensure that the previous questions and answers do not influence the current question.

Static analysis tool support. As mentioned in §4.3, we utilized ANTLR [21] to parse the Solidity source code and generate an abstract syntax tree (AST). ANTLR allows for source code analysis without the need for compilation, making it more effective for source code with limited dependencies and build scripts compared to tools relying on compilation, such as Slither [37]. Furthermore, to determine data dependencies between two variables or expressions in §4.4, we employed a static analysis tool [23] based on the output of crytic-compiler [7], a Solidity compiler capable of producing a standard AST for static analysis. With this approach, we can construct both a control flow graph and a data dependence graph.

5 EVALUATION

In this section, we conduct experiments to evaluate GPTScan's accuracy, performance, financial overhead, the effectiveness of its static confirmation, and its capability to discover new vulnerabilities.

Datasets. As shown in Table 2, the experiments were conducted on three datasets collected from real-world smart contracts. These datasets consist of around 400 contract projects, 3K Solidity files, 472K lines of code, and include 62 ground-truth logic vulnerabilities.

Table 2: Three diverse datasets for GPTScan's evaluation.

Dataset Name	Projects P	Files F	F/P	LoC	Vuls
Top200	303	555	1.83	134,322	0
Web3Bugs	72	2,573	35.74	319,878	48
DefiHacks	13	29	2.23	17,824	14
Sum	388	3,157	8.14	472,024	62

The first dataset, called *Top200*, comprises smart contracts with a top 200 market capitalization. It includes 303 open-source contract projects from six mainstream Ethereum-compatible chains [62]. Since these projects are well-audited and widely used, it is reasonable to assume that they do not contain notable vulnerabilities. This dataset is primarily used to stress-test the false-positive rate of GPTScan in audited contracts. The second dataset, called *Web3Bugs*,, was collected from the recent Web3Bugs dataset [8, 65], which comprises 100 Code4rena-audited projects. Among the 100 projects, we included 72 projects that can be directly compiled. The remaining projects either miss library dependencies or configuration files in their original Web3Bugs repository [8]. The third dataset, called *DefiHacks*, come from the well-known DeFi Hacks dataset [9], which consists of vulnerable token contracts that have incurred past attack incidents. We included 13 vulnerable projects that certainly cover the vulnerabilities in our ten types. The ground-truth vulnerabilities in these datasets include those already reported and those newly detected by GPTScan and confirmed by the community.

All these projects are compiled with crytic-compiler [7] using the default configuration. Note that 17 projects in the *Top200* dataset cannot be compiled with crytic-compiler. For these projects, GPTScan's static confirmation cannot be applied, and any influenced types of vulnerabilities will be marked as not detected.

Research Questions. With the datasets above, we aim to answer the following five research questions (RQs):

RQ1: What is the false positive rate of GPTScan when analyzing a dataset of non-vulnerable top contracts?
RQ2: How accurate is GPTScan in analyzing real-word datasets with logic vulnerabilities, and how effective is it compared to existing tools?
RQ3: How effective is GPTScan's static confirmation in improving the accuracy of GPTScan?
RQ4: What are the running performance and financial costs of GPTScan?
RQ5: Can GPTScan discover new vulnerabilities that were previously missed by human auditors?

5.1 RQ1: Measuring False Positives in the Non-vulnerable Top Contracts

In RQ1, we aim to measure GPTScan's false alarm rate in analyzing non-vulnerable contracts. This is important because when using GPTScan for massive scanning of on-chain token contracts, we want to minimize the false alarms that require manual checking.

For this purpose, we have collected the *Top200* dataset, which consists of 303 contract projects that are deemed non-vulnerable. We present GPTScan's analysis result of *Top200* in Table 3. Along with the results of *Web3Bugs* and *DefiHacks*, we calculate the accuracy metrics at the function level for each tested vulnerability type. For example, if a project has been tested with five vulnerability

Table 3: Overall results of GPTScan's accuracy evaluation.

Dataset Name	TP	TN	FP	FN	Sum
Top200	0	283	13	0	296
Web3Bugs	40	154	30	8	232
DefiHacks	10	19	1	4	34

types, the sum of all true positives, false positives, true negatives, and false negatives for this project should be 5. More specifically,
TP is the number of true positives. One true positive is counted when GPTScan successfully detects a ground-truth vulnerable function for the tested vulnerability type.
TN is the number of true negatives. One true negative is counted when GPTScan correctly does not report any vulnerable function for the tested vulnerability type.
FP is the number of false positives. One false positive is counted when GPTScan incorrectly reports one or more vulnerable functions for the tested vulnerability type that has no corresponding ground-truth vulnerabilities in the tested project.
FN is the number of false negatives. One false negative is counted when GPTScan fails to detect the ground-truth vulnerable function for the tested vulnerability type.

Based on the calculation of these metrics, GPTScan reports 13 FPs and 283 TNs for the *Top200* dataset, as shown in Table 3. As a result, the false positive rate of GPTScan in analyzing non-vulnerable top contracts like *Top200* is 4.39%. Moreover, we find that GPTScan has a similar precision when analyzing *Top200* and *DefiHacks*, both of which are token contracts with around 2 Solidity files per project (see Table 2). When analyzing large projects like those in *Web3Bugs*, the precision drops from around 90% (90.91% for *DefiHacks*) to 60% (57.14% for *Web3Bugs*). The drop in precision is likely because the smart contract code in *Web3Bugs* is more diverse, given that *Web3Bugs* contains an average of 36 Solidity files per project (see Table 2). In contrast, smart contracts in *DefiHacks* and *Top200* mainly implement common token functionalities using an average of 2 Solidity files per project, potentially triggering only a limited set of false positives in GPTScan. In §5.2, we will further discuss the root causes of GPTScan's false positives.

Answer for RQ1: GPTScan achieves a low false positive rate of 4.39% when analyzing non-vulnerable top contracts like *Top200*. It also demonstrates similar performance in analyzing *DefiHacks*, with a precision of 90.91%. These results indicate that GPTScan is suitable for massive scanning of on-chain token contracts. Moreover, when analyzing large contract projects in *Web3Bugs*, GPTScan still achieves an acceptable precision of 57.14%.

5.2 RQ2: Efficacy for Detecting Vulnerable Contracts

In RQ2, we assess the effectiveness of GPTScan in analyzing vulnerable contracts in the *Web3Bugs* and *DefiHacks* datasets, and compare its effectiveness with existing tools.

As shown in Table 2, the Web3Bugs dataset contains 48 ground-truth logic vulnerabilities, while the *DefiHacks* dataset has 14. Table 3 presents the scanning results of these two datasets using GPTScan. In the case of *Web3Bugs*, GPTScan analyzed a total of 232 vulnerability types across 72 projects, detecting 40 TPs and missing 8 FNs, while incurring 30 FPs. Consequently, GPTScan achieved a recall of 83.33% and an F1 score of 67.8% on this dataset. For *DefiHacks*, GPTScan analyzed a total of 34 vulnerability types across 13 projects, detecting 10 TPs and missing 4 FNs, while incurring 1 FP. On this dataset, GPTScan's recall is 71.43% and the F1 score is 80%. These results demonstrate that GPTScan effectively detects vulnerable contracts for the covered logic vulnerability types. Following the initial precision analysis in §5.1, we now analyze the root causes of GPTScan's false negatives and false positives.

In the 12 false negative cases, 4 of them are *Price Manipulation by AMM* and 3 of them are *Risky First Deposit*. The main reason for these two kinds of false negatives is that GPTScan does not implement an alias analysis in the static check, causing failure during static dataflow tracing. Additionally, there are 2 cases of *Front Running*, where the scenarios or properties are not accurately matched by GPT. Furthermore, there are 2 cases of *Slippage* and 1 case of *Unauthorized Transfer*. Similar to the false positive cases, The main reason for the false negative *Slippage* cases is the existence of numerous variants of slippage checks, making them challenging to detect using GPT and static analysis. In the case of *Unauthorized Transfer*, the main reason for this false negative is that GPT failed to distinguish the inconsistency between the comment and code.

GPTScan achieves effective vulnerability detection above at an acceptable false alarm rate. Among the 44 false positive cases from the three datasets, 15 (34.09%) were related to *Price Manipulation by AMM*, followed by 11 (25.00%) cases of *Unauthorized Transfer*. For these two types, the main reason for the false alarms is that these vulnerabilities require specific triggering conditions involving other related logic, which may not be contained within a single function and its callers or callees. For example, in *Unauthorized Transfer*, the checks for the allowance/approval from the address owner can occur at various positions in the logic chain and may involve multiple functions. Similarly, the function that calculates the price with AMM for *Price Manipulation* may not be used by other functions responsible for swapping or buying tokens, leading to the vulnerabilities not being triggered in those circumstances.

Additionally, there were 5 cases of *Risky First Deposit* and 5 cases of *Slippage*. For *Risky First Deposit*, the false alarms occurred because there were many statements related to checking the supply and setting the share, making it challenging for GPT to understand lengthy code segments accurately. Regarding *Slippage*, the false alarms were mainly due to two factors. First, similar to *Unauthorized Transfer*, the check for slippage can happen at any position in the logic chain, and second, slippage checks can take many different forms and variants, making them difficult to detect with GPT and static analysis. For this vulnerability type, our focus was on achieving a higher recall at the cost of slightly sacrificing precision. There were also 4 cases of *Wrong Interest Rate Order*, 3 cases of *Approval Not Cleared*, and 1 case of *Wrong Checkpoint Order*. For *Wrong Interest Rate Order* and *Wrong Checkpoint Order*, these vulnerabilities are intricately related to the business logic of the project itself, making it challenging to reduce false alarms without comprehensive knowledge of the project's design. As for *Approval Not Cleared*, the false alarms were primarily because the function may not always be used to transfer tokens, causing GPTScan to detect it erroneously.

Comparison with existing tools. While there are many specific static analysis tools (e.g., [28, 29, 47, 56]), they almost do not cover any of the logic vulnerabilities targeted in this paper. We thus selected two comprehensive vulnerability detection tools, one open-source tool, Slither [37], and MetaScan's online static scanning service [19, 23], referred to as MScan. Both tools have over a hundred vulnerability detection rules, but the rules related to GPTScan are *unchecked-transfer*, *arbitrary-send-eth*, and *arbitrary-send-erc20* for Slither (corresponding to *Unauthorized Transfer* in GPTScan), and two *Price Manipulation* vulnerabilities for MScan.

We ran Slither on all three datasets and found a total of 13,144 warnings. Among these, only 101 of *unchecked-transfer*, 23 of *arbitrary-send-eth*, and 22 of *arbitrary-send-erc20* are related to the *Unauthorized Transfer* vulnerability in GPTScan. Unfortunately, all of them were false positives after careful manual checking. There are mainly two reasons for this. Firstly, Slither does not correlate call chain information. Many false positive cases involve `internal` or `private` functions that have already been checked for unauthorized transfer when they are called. In GPTScan, we analyze the current function and its caller together, effectively addressing the issue of missing contextual semantics. Secondly, Slither is unable to correctly detect variants of transfer behavior in *Unauthorized Transfer*, such as burning tokens, leading to its inability to detect vulnerabilities in the dataset. GPTScan relies on GPT to gain the ability to analyze code semantics, which, when combined with code context and calling relationships, can more accurately address these problems.

We also ran MScan on the *DefiHacks* dataset, as 12 of the total 14 vulnerabilities in this dataset are related to *Price Manipulation*. Among these 12 true *Price Manipulation* vulnerabilities, MScan detected 7, achieving a recall of 58.33% and a precision of 100% for *Price Manipulation*. However, MScan failed to detect any other type of logic vulnerabilities. MScan achieved high precision because it used some attack incidents in the *DefiHacks* dataset to derive hard-coded patterns for *Price Manipulation*, including the matching of specific function and variable names. However, in cases where hard-coded patterns are not applicable, MScan cannot generalize to detect variants of *Price Manipulation* vulnerabilities.

For GPT-based tools, the only available study at the time of our submission was conducted by David et al. [34]. Unfortunately, they did not release their tool, and there was insufficient information for us to reproduce it. Therefore, we rely on the statistics provided in their paper for comparison. According to the paper, their pure GPT-based approach achieved a precision of 4.14%, a recall of 43.84%, and an F1 score of 7.57% with the GPT-4-32k model, and a precision of 4.30%, a recall of 35.62%, and an F1 score of 7.68% with the Claude-v1.3-100k model, respectively. The false positives are significantly higher than those of GPTScan, mainly because their tool did not validate the GPT output as GPTScan does in §4.4, and thus could be more easily affected by GPT's inherent problems like hallucination [64], bias in training data, and ambiguity in questions. Indeed, RQ3 in §5.3 suggests a similar finding by measuring the GPT-only result in GPTScan (see details in Table 4).

Answer for RQ2: GPTScan shows its efficacy in detecting ground-truth logic vulnerabilities in the *Web3Bugs* and *DefiHacks* datasets, with a recall of 83.33% and an F1 score of 67.8% for *Web3Bugs*, and a recall of 71.43% and an F1 score of 80% for *DefiHacks*, better than existing static and GPT-based tools.

Table 4: Raw functions before and after static confirmation.

Vulnerability Type	Before	After
Approval Not Cleared	34	12
Risky First Deposit	100	21
Price Manipulation by AMM	187	114
Price Manipulation by Buying Tokens	8	8
Vote Manipulation by Flashloan	2	0
Front Running	6	4
Wrong Interest Rate Order	150	11
Wrong Checkpoint Order	49	1
Slippage	99	42
Unauthorized Transfer	12	8
Total	**647**	**221**

5.3 RQ3: Effectiveness of Static Confirmation

In RQ3, we conduct a further analysis of GPTScan's intermediate results on *Web3Bugs* to examine how static confirmation reduces false positives generated by pure GPT-based matching.

Table 4 shows the raw functions reported by GPTScan before and after static confirmation. Note that one vulnerability type may have multiple functions (the final result counts either TP or FP once, according to the calculation in §5.1), and these functions are not merged yet (i.e., a function A and the combination of function A and all its callers would be counted multiple times) that will be done in the final result. Hence, so the number of "after" cases shown here is much larger than the final TP+FP in Table 3. From the result, we observe that static confirmation effectively filters out most false positive cases for the vulnerability types: *Wrong Interest Rate Order*, *Wrong Checkpoint Order* and *Risky First Deposit*. The reason is that the description of scenarios and properties for these three types is coarse-grained, leading to many candidate functions passing the GPT-based matching step. In static confirmation, GPTScan can further instruct GPT to identify related statements and variables, filtering out those that do not satisfy the vulnerability types. Overall, after static confirmation, only 221 raw functions remain out of the original 647 functions. This indicates that static confirmation successfully filters out two-thirds of the false positives.

We further analyze the negative impact of static confirmation. Among the 426 cases filtered out, only 3 ground-truth cases were initially matched by GPT but later excluded by static analysis, resulting in 3 false negatives. Another false negative was related to compilation problems. The remaining four did not pass the GPT-based scenario and property matching step. This indicates that static confirmation has only a minor impact on the false negatives.

Answer for RQ3: Static confirmation effectively filtered out 65.84% of the false positive cases in the *Web3Bugs* dataset, while having only a minor impact on the false negative cases.

5.4 RQ4: Performance and Financial Overhead

In RQ4, we evaluate the running time and financial costs of GPTScan when using OpenAI's GPT-3.5-turbo API. We considered only the costs associated with interacting with GPT and conducting static analysis. We measured the time and financial cost of GPTScan on all three datasets, and the results are shown in Table 5. In this experiment, we used tiktoken [20], a tokenization tool published by OpenAI and used for GPT models, to estimate the number of tokens

Table 5: Running time and financial costs of GPTScan.

Dataset	KL*	T**	C***	T/KL	C/KL
Top200	134.32	1,437.37	0.7507	10.70	0.005589
Web3Bugs	319.88	4,980.57	3.9682	15.57	0.018658
DefiHacks	17.82	375.41	0.2727	21.06	0.015303
Overall	**472.02**	**6,793.35**	**4.9984**	**14.39**	**0.010589**

* KL for KLoC; ** T for Time; *** C for Financial Cost.

sent and received by GPTScan. With the number of tokens sent and received, we can estimate the financial cost of GPTScan. The total number of lines of code is 472K, and it took 6,793.35 seconds and 4.9984 USD to complete the scan. On average, it takes 14.39 seconds and 0.010589 USD to scan per thousand lines of code.

On *Top200*, the scan cost per thousand lines of code is the cheapest, and the scan speed per thousand lines of code is the fastest. This is because most candidate functions are filtered out in GPTScan's first two steps, without the need for finding related variables and expressions for static check. On *Web3Bugs* and *DefiHacks*, the scan cost per thousand lines of code is the most expensive and the scan speed per thousand lines of code is the slowest, respectively. Projects in *Web3Bugs* and *DefiHacks* are more complex than *Top200*, and there are more complex candidate functions to be scanned. These complex functions could not be filtered by static filtering and scenario matching, which causes more time and financial cost.

Answer for RQ4: GPTScan is fast and cost-effective, taking an average of only 14.39 seconds and 0.01 USD to scan per thousand lines of Solidity code in the tested datasets. The relatively higher cost and slower speed for *Web3Bugs* and *DefiHacks* can be attributed to the presence of more complex functions that cannot be filtered out by static filtering and scenario matching.

5.5 RQ5: Newly Discovered Vulnerabilities

In RQ5, we perform a thorough analysis of GPTScan's results on the *Web3Bugs* dataset to see if it could identify new vulnerabilities that were previously missed by human auditors. Interestingly, GPTScan successfully discovered 9 vulnerabilities from 3 different types, which did not appear in the audit reports of Code4rena. Among these 9 newly discovered vulnerabilities, 5 are *Risky First Deposit*, 3 are *Price Manipulation by AMM*, and 1 is *Front Running*. In the following paragraphs, we present one example of each type of newly discovered vulnerability for further discussion.

Risky First Deposit. Among the newly discovered vulnerabilities, 56% of them are *Risky First Deposit*. In the example shown in Figure 6, on line 11, when the variable `_pool` is 0, indicating an empty liquidity pool, the depositor can obtain all the shares from the pool. The presence of both `_totalSupply` and `_pool` variables to represent the liquidity amount in the pool may confuse human auditors. Although lines 5 to 8 properly handle the case when `_totalSupply` is 0, this specific condition involving `_pool` on line 11 creates a vulnerability that could be missed.

Price Manipulation by AMM. Another 33% of the newly discovered vulnerabilities are *Price Manipulation by AMM*. In the example shown in Figure 7, the `pendingRewards` function is used to calculate the rewards that can be claimed by the user. On line 9, when the pool is not empty, the amount of rewards that can be redeemed by the user is calculated based on the total supply in the pool. However, the total supply can be controlled by users, allowing them to manipulate the redeemed amount and exploit the contracts.

```
1  function deposit(uint _amount) external {
2      ...
3      uint _pool = balance();
4      uint _totalSupply = totalSupply();
5      if (_totalSupply == 0 && _pool > 0) { // trading fee
           accumulated while there were no IF LPs
6          vusd.safeTransfer(governance, _pool);
7          _pool = 0;
8      }
9      uint shares = 0;
10     if (_pool == 0) {
11         shares = _amount;
12     } else {
13         shares = _amount * _totalSupply / _pool;
14     }
15     ...
16 }
```

Figure 6: *Risky First Deposit* in 2022-02-hubble.

```
1  function pendingRewards(uint256 _pid, address _user) external
        view returns (uint256) {
2      PoolInfo storage pool = poolInfo[_pid];
3      UserInfo storage user = userInfo[_pid][_user];
4      uint256 accRewardsPerShare = pool.accRewardsPerShare;
5      uint256 lpSupply = pool.lpToken.balanceOf(address(this));
6      if (block.number > pool.lastRewardBlock && lpSupply != 0) {
7          uint256 multiplier = getMultiplier(pool.lastRewardBlock
               , block.number);
8          uint256 rewardsAccum = multiplier.mul(rewardsPerBlock).
               mul(pool.allocPoint).div(totalAllocPoint);
9          accRewardsPerShare = accRewardsPerShare.add(
               rewardsAccum.mul(1e12).div(lpSupply));
10     }
11     return user.amount.mul(accRewardsPerShare).div(1e12).sub(
           user.rewardDebt);
12 }
```

Figure 7: *Price Manipulation by AMM* in 2021-09-sushimiso.

```
1  /// @notice The lp tokens that the user contributes need to
        have been transferred previously, using a batchable
        router.
2  function mint(address to)
3      public
4      beforeMaturity
5      returns (uint256 minted)
6  {
7      uint256 deposit = pool.balanceOf(address(this)) - cached;
8      minted = _totalSupply * deposit / cached;
9      cached += deposit;
10     _mint(to, minted);
11 }
```

Figure 8: *Front Running* in 2021-08-yield.

Front Running. There is one case of *Front Running* shown in Figure 8, in which the token to be minted should be previously transferred (line 1). However, anyone can call the `mint` function to mint tokens that are transferred but not minted, as there is only a check with the cached amount of the contract (line 7), but not the cached amount of a specific user. This vulnerability allows an attacker to front run the minting process. When a user has transferred a token but not minted it, the attacker could front run the `mint` function to mint the token before the legitimate user.

Answer for RQ5: GPTScan identified 9 new vulnerabilities not present in the audit reports of Code4rena. This highlights the value of GPTScan as a useful supplement to human auditors.

6 DISCUSSION

In this section, we discuss the current limitations in GPTScan and the potential use of employing other GPT models.

Current limitations in design and implementation. In §4.3, the modifiers filtering part only utilized a whitelist to filter the modifiers with access control. However, this filtering method can lead to false positives or negatives of vulnerabilities. To enhance accuracy, a more precise approach is required, which involves retrieving the definition of modifiers and conducting a detailed semantic analysis on them. For the static analysis part in §4.4, a simple method was used to analyze the control flow graph and data dependence graph. This analysis is not path-sensitive, meaning that some path-related issues, such as the reachability of certain execution paths under specific conditions, might be overlooked. It could be improved by introducing symbolic execution engines to the static analysis part.

The use of other GPT models and parameters. As mentioned in §4.5, GPTScan employs the widely used GPT-3.5-turbo model [27] as its GPT model. We also conducted a preliminary test using GPT-4, but we did not observe a notable improvement, while the cost increased 20 times. This finding suggests that GPTScan does not necessarily require more powerful GPT models. As the temperature parameter is set to zero, the answers of the GPT model tend to be deterministic. A higher temperature might lead to more creative answers, but it could also result in more false positives or false negatives. However, reproducing results becomes more challenging with a higher temperature. In the future, we plan to conduct a systematic test of various GPT models within the context of GPTScan, including Google Bard, Claude (when we have API access to them), and the self-trained LLaMA model, as well as the influence of different parameters on GPTScan.

7 RELATED WORK

In this section, we discuss some related work. Various research and tools have focused on vulnerability detection in smart contracts. Traditional static analysis tools, such as Slither [37], Vandal [30], Ethainter [29], Zues [43], and Securify [56], are used to analyze the source code and detect vulnerabilities. Symbolic execution tools like Manticore [47] and Mythril [13] can perform bound checks and detect vulnerabilities in bytecode and source code. These analysis tools have been applied to detect vulnerabilities in smart contracts, such as re-entrancy [52, 61], arithmetic overflow [54], state inconsistency problems [28], and access control problems [36, 39, 46]. Dynamic analysis tools, such as fuzz testing [40, 41, 59, 63], automatically generate test cases or inputs for smart contracts to find abnormal behaviors during runtime. Formal verification techniques like Verx [51] and VeriSmart [53] can be used to check user-provided specifications. Nevertheless, Zhang et al. [65] suggested that more than 80% of exploitable bugs are machine undetectable.

Before the advent of ChatGPT (GPT-3.5) [49], most NLP-based vulnerability detection methods [32, 33, 48, 55, 58] involved feeding code into binary or multi-classification models. Now, with the development of instructing GPT [57] and other research providing few-shot learning capabilities [31], interactive solutions can be used for tasks like code repair [42, 60] and vulnerability detection [34]. However, according to the research by David et al. [34], the GPT-4 model itself cannot accurately detect vulnerabilities. Chen et al. [38] fine-tuned the GPT-3 model for improved performance in GUI graphical interface testing tasks and utilized it for automated testing of Android applications. Additionally, PentestGPT [24] and ChatRepair [60] utilized feedback from the execution results to enhance the performance of the GPT model during interactions.

8 CONCLUSION

In this paper, we proposed GPTScan, the first tool combining GPT with static analysis for smart contract logic vulnerability detection. GPTScan utilized GPT to match candidate vulnerable functions based on code-level scenarios and properties, and further instructed GPT to intelligently recognize key variables and statements, which were then validated by static confirmation. Our evaluation on three diverse datasets with around 400 contract projects and 3K Solidity files showed that GPTScan achieves high precision (over 90%) for token contracts and acceptable precision (57.14%) for large projects, as well as a recall of over 70% for detecting ground-truth logic vulnerabilities. GPTScan is fast, cost-effective, and capable of discovering new vulnerabilities missed by human auditors. In future work, we will expand GPTScan's support for more logic vulnerability types.

ACKNOWLEDGEMENTS

We thank Dawei Zhou, Zhe Wang, Guorui Fan, Liwei Tan, Hao Zhang, and other colleagues at MetaTrust Labs for their help with GPTScan, as well as anonymous reviewers for their constructive feedback. This research/project is supported by the National Research Foundation Singapore and DSO National Laboratories under the AI Singapore Programme (AISG Award No: AISG2-RP-2020-019), the National Research Foundation, Singapore, and the Cyber Security Agency under its National Cybersecurity R&D Programme (NCRP25-P04-TAICeN). Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the views of National Research Foundation, Singapore and Cyber Security Agency of Singapore.

REFERENCES

[1] 2016. https://www.coindesk.com/learn/understanding-the-dao-attack/
[2] 2021. https://github.com/code-423n4/2021-11-yaxis
[3] 2022. https://www.freecodecamp.org/news/what-is-yaml-the-yml-file-format/
[4] 2023. https://blog.openzeppelin.com/on-the-parity-wallet-multisig-hack-405a8c12e8f7
[5] 2023. https://openai.com/chatgpt
[6] 2023. https://openai.com/pricing
[7] 2023. https://github.com/crytic/crytic-compile
[8] 2023. https://github.com/ZhangZhuoSJTU/Web3Bugs
[9] 2023. https://wooded-meter-1d8.notion.site/0e85e02c5ed34df3855ea9f3ca40f53b?v=22e5e2c506ef4caeb40b4f78e23517ee
[10] 2023. https://code4rena.com/
[11] 2023. https://soliditylang.org/
[12] 2023. https://docs.soliditylang.org/en/latest/smtchecker.html
[13] 2023. https://github.com/Consensys/mythril
[14] 2023. https://blog.trailofbits.com/2023/03/22/codex-and-gpt4-cant-beat-humans-on-smart-contract-audits/
[15] 2023. https://github.com/code-423n4/2022-04-jpegd-findings/issues/12
[16] 2023. https://github.com/code-423n4/2022-04-backd
[17] 2023. https://github.com/code-423n4/2022-04-backd-findings/issues/36
[18] 2023. https://github.com/code-423n4/2022-05-backd/blob/2a5664d35cde5b036074edef3c1369b984d10010/protocol/contracts/StakerVault.sol
[19] 2023. https://app.metatrust.io/
[20] 2023. https://github.com/openai/tiktoken
[21] 2023. ANTLR. https://www.antlr.org/

[22] 2023. Breaking Barriers: GPTScan's Game-changing Role in Smart Contract Security. https://metatrust.io/company/newsroom/post/breaking-barriers-gptscans-gamechanging-role-in-smart-contract-security

[23] 2023. falcon-metatrust: MetaTrust fork of Slither Analyzer. https://github.com/MetaTrustLabs/falcon-metatrust

[24] 2023. GreyDGL/PentestGPT. https://github.com/GreyDGL/PentestGPT

[25] 2023. MetaScan v1.6: Unparalleled Visibility and AI Security for Smart Contracts. https://metatrust.io/company/newsroom/post/metascan-v16-unparalleled-visibility-and-ai-security-for-smart-contracts

[26] 2023. OpenZeppelin. https://www.openzeppelin.com

[27] 2023. Overview - OpenAI API. https://platform.openai.com

[28] 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.

[29] Lexi Brent, Neville Grech, Sifis Lagouvardos, Bernhard Scholz, and Yannis Smaragdakis. 2020. Ethainter: a smart contract security analyzer for composite vulnerabilities. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. 454–469.

[30] Lexi Brent, Anton Jurisevic, Michael Kong, Eric Liu, Francois Gauthier, Vincent Gramoli, Ralph Holz, and Bernhard Scholz. 2018. Vandal: A scalable security analysis framework for smart contracts. *arXiv preprint arXiv:1809.03981* (2018).

[31] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems* 33 (2020), 1877–1901.

[32] Yizheng Chen, Zhoujie Ding, Xinyun Chen, and David Wagner. 2023. DiverseVul: A New Vulnerable Source Code Dataset for Deep Learning Based Vulnerability Detection. *arXiv preprint arXiv:2304.00409* (2023).

[33] Anton Cheshkov, Pavel Zadorozhny, and Rodion Levichev. 2023. Evaluation of ChatGPT Model for Vulnerability Detection. *arXiv preprint arXiv:2304.07232* (2023).

[34] 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 (Jun 2023). http://arxiv.org/abs/2306.12338 arXiv:2306.12338 [cs].

[35] Yinlin Deng, Chunqiu Steven Xia, Haoran Peng, Chenyuan Yang, and Lingming Zhang. 2023. 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*. ACM, Seattle WA USA, 423–435. https://doi.org/10.1145/3597926.3598067

[36] Yuzhou Fang, Daoyuan Wu, Xiao Yi, Shuai Wang, Yufan Chen, Mengjie Chen, Yang Liu, and Lingxiao Jiang. 2023. Beyond "Protected" and "Private": An Empirical Security Analysis of Custom Function Modifiers in Smart Contracts. In *Proc. ACM ISSTA*.

[37] 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.

[38] Sidong Feng and Chunyang Chen. 2023. Prompting Is All Your Need: Automated Android Bug Replay with Large Language Models. *arXiv preprint arXiv:2306.01987* (2023).

[39] Asem Ghaleb, Julia Rubin, and Karthik Pattabiraman. 2023. AChecker: Statically Detecting Smart Contract Access Control Vulnerabilities. *Proc. ACM ICSE* (2023).

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

[41] 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.

[42] Nan Jiang, Kevin Liu, Thibaud Lutellier, and Lin Tan. 2023. Impact of code language models on automated program repair. *arXiv preprint arXiv:2302.05020* (2023).

[43] Sukrit Kalra, Seep Goel, Mohan Dhawan, and Subodh Sharma. 2018. ZEUS: Analyzing Safety of Smart Contracts. In *Proceedings 2018 Network and Distributed System Security Symposium*. Internet Society, San Diego, CA.

[44] Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. 2022. Large language models are zero-shot reasoners. *Advances in neural information processing systems* 35 (2022), 22199–22213.

[45] Emi Lacapra. 2023. What are liquidity provider (LP) tokens, and how do they work? https://cointelegraph.com/explained/what-are-liquidity-provider-lp-tokens-and-how-do-they-work

[46] Ye Liu, Yi Li, Shang-Wei Lin, and Cyrille Artho. 2022. Finding permission bugs in smart contracts with role mining. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*. 716–727.

[47] Mark Mossberg, Felipe Manzano, Eric Hennenfent, Alex Groce, Gustavo Greico, Josselin Feist, Trent Brunson, and Artem Dinaburg. 2019. Manticore: A User-Friendly Symbolic Execution Framework for Binaries and Smart Contracts. In *Proc. ACM ASE*.

[48] Marwan Omar. 2023. Detecting software vulnerabilities using Language Models. *arXiv preprint arXiv:2302.11773* (2023).

[49] Long Ouyang, Jeff Wu, Xu Jiang, Diogo Almeida, Carroll L. Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, John Schulman, Jacob Hilton, Fraser Kelton, Luke Miller, Maddie Simens, Amanda Askell, Peter Welinder, Paul Christiano, Jan Leike, and Ryan Lowe. 2022. Training language models to follow instructions with human feedback. https://doi.org/10.48550/arXiv.2203.02155 arXiv:2203.02155 [cs].

[50] Long Ouyang, Jeff Wu, Xu Jiang, Diogo Almeida, Carroll L. Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, John Schulman, Jacob Hilton, Fraser Kelton, Luke Miller, Maddie Simens, Amanda Askell, Peter Welinder, Paul Christiano, Jan Leike, and Ryan Lowe. 2022. Training language models to follow instructions with human feedback. (2022). https://doi.org/10.48550/ARXIV.2203.02155

[51] Anton Permenev, Dimitar Dimitrov, Petar Tsankov, Dana Drachsler-Cohen, and Martin Vechev. 2020. Verx: Safety verification of smart contracts. In *2020 IEEE symposium on security and privacy (SP)*. IEEE, 1661–1677.

[52] Michael Rodler, Wenting Li, Ghassan O. Karame, and Lucas Davi. 2019. Sereum: Protecting Existing Smart Contracts Against Re-Entrancy Attacks. In *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*. The Internet Society.

[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] Bryan Tan, Benjamin Mariano, Shuvendu K Lahiri, Isil Dillig, and Yu Feng. 2022. SolType: refinement types for arithmetic overflow in solidity. In *Proc. ACM POPL*.

[55] Chandra Thapa, Seung Ick Jang, Muhammad Ejaz Ahmed, Seyit Camtepe, Josef Pieprzyk, and Surya Nepal. 2022. Transformer-based language models for software vulnerability detection. In *Proceedings of the 38th Annual Computer Security Applications Conference*. 481–496.

[56] 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.

[57] Yizhong Wang, Yeganeh Kordi, Swaroop Mishra, Alisa Liu, Noah A. Smith, Daniel Khashabi, and Hannaneh Hajishirzi. 2022. Self-Instruct: Aligning Language Model with Self Generated Instructions.

[58] Hongjun Wu, Zhuo Zhang, Shangwen Wang, Yan Lei, Bo Lin, Yihao Qin, Haoyu Zhang, and Xiaoguang Mao. 2021. Peculiar: Smart contract vulnerability detection based on crucial data flow graph and pre-training techniques. In *2021 IEEE 32nd International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 378–389.

[59] Valentin Wüstholz 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*. 1398–1409.

[60] Chunqiu Steven Xia and Lingming Zhang. 2023. Keep the Conversation Going: Fixing 162 out of 337 bugs for $0.42 each using ChatGPT. *arXiv preprint arXiv:2304.00385* (2023).

[61] Yinxing Xue, Mingliang Ma, Yun Lin, Yulei Sui, Jiaming Ye, and Tianyong Peng. 2020. Cross-Contract Static Analysis for Detecting Practical Reentrancy Vulnerabilities in Smart Contracts. In *35th IEEE/ACM International Conference on Automated Software Engineering, ASE 2020, Melbourne, Australia, September 21-25, 2020*. IEEE, 1029–1040.

[62] Xiao Yi, Yuzhou Fang, Daoyuan Wu, and Lingxiao Jiang. 2023. BlockScope: Detecting and Investigating Propagated Vulnerabilities in Forked Blockchain Projects. In *Proc. ISOC NDSS*.

[63] William Zhang, Sebastian Banescu, Leonardo Pasos, Steven Stewart, and Vijay Ganesh. 2019. Mpro: Combining static and symbolic analysis for scalable testing of smart contract. In *2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 456–462.

[64] Yue Zhang, Yafu Li, Leyang Cui, Deng Cai, Lemao Liu, Tingchen Fu, Xinting Huang, Enbo Zhao, Yu Zhang, Yulong Chen, Longyue Wang, Anh Tuan Luu, Wei Bi, Freda Shi, and Shuming Shi. 2023. Siren's Song in the AI Ocean: A Survey on Hallucination in Large Language Models. arXiv:2309.01219 [cs.CL]

[65] Zhuo Zhang, Brian Zhang, Wen Xu, and Zhiqiang Lin. 2023. Demystifying Exploitable Bugs in Smart Contracts. In *2023 IEEE/ACM 37th IEEE International Conference on Software Engineering*.

[66] Liyi Zhou, Xihan Xiong, Jens Ernstberger, Stefanos Chaliasos, Zhipeng Wang, Ye Wang, Kaihua Qin, Roger Wattenhofer, Dawn Song, and Arthur Gervais. 2023. SoK: Decentralized Finance (DeFi) Attacks. In *IEEE Symposium on Security and Privacy (SP)*. IEEE.