DeFiAligner: Leveraging Symbolic Analysis and Large Language Models for Inconsistency Detection in Decentralized Finance

Rundong Gan
School of Computer Science, University of Guelph, Canada

Liyi Zhou
The University of Sydney, Australia
UC Berkeley RDI, CA, USA
Decentralized Intelligence AG, Zug, Switzerland

Le Wang
School of Computer Science, University of Guelph, Canada

Kaihua Qin
Yale University, New Haven, CT, USA
UC Berkeley RDI, CA, USA
Decentralized Intelligence AG, Zug, Switzerland

Xiaodong Lin[1]
School of Computer Science, University of Guelph, Canada

Abstract

Decentralized Finance (DeFi) has witnessed a monumental surge, reaching 53.039 billion USD in total value locked. As this sector continues to expand, ensuring the reliability of DeFi smart contracts becomes increasingly crucial. While some users are adept at reading code or the compiled bytecode to understand smart contracts, many rely on documentation. Therefore, discrepancies between the documentation and the deployed code can pose significant risks, whether these discrepancies are due to errors or intentional fraud. To tackle these challenges, we developed *DeFiAligner*, an end-to-end system to identify inconsistencies between documentation and smart contracts. *DeFiAligner* incorporates a symbolic execution tool, SEVM, which explores execution paths of on-chain binary code, recording memory and stack states. It automatically generates symbolic expressions for token balance changes and branch conditions, which, along with related project documents, are processed by LLMs. Using structured prompts, the LLMs evaluate the alignment between the symbolic expressions and the documentation. Our tests across three distinct scenarios demonstrate *DeFiAligner* 's capability to automate inconsistency detection in DeFi, achieving recall rates of 92% and 90% on two public datasets respectively.

2012 ACM Subject Classification Security and privacy → Distributed systems security

Keywords and phrases Decentralized Finance Security, Large Language Models, Project Review, Symbolic Analysis, Smart Contracts

Digital Object Identifier 10.4230/LIPIcs.AFT.2024.7

Supplementary Material *Software (Source Code)*: `https://github.com/DeFiAligner/DeFiAligner` [28], archived at `swh:1:dir:ceb930c5854a5cae98402c9cd310fdb2c940ceeb`

Funding This work was supported in part by the National Sciences and Engineering Research Council of Canada (NSERC).

[1] Corresponding author

1 Introduction

Decentralized Finance (DeFi) encompasses a wide range of financial applications and services built on blockchain platforms that support smart contracts, such as exchanges, lending and borrowing platforms, and derivatives [54]. These smart contracts are pieces of code that execute automatically when triggered by transactions. A distinct characteristic of DeFi is the transparency of the compiled bytecode for all smart contracts, often encapsulated by the maxim *"code is law"*. This principle underscores the ecosystem's transparency, allowing anyone to deterministically and independently verify state transitions and validate the execution outcomes of these contracts and transactions.

Although the DeFi ecosystem offers sufficient transparency to allow anyone to verify and review the code of smart contracts, discrepancies between project documentation and the actual code can still pose risks to users. DeFi project documentation typically describes core functionalities and financial models, yet the actual on-chain code (e.g., EVM bytecode) may not faithfully implement these features due to programming errors or intentional design, and may even include unmentioned functionalities. A typical example is the Uranium Finance incident [5], where a discrepancy between the implementation of a conditional formula in the deployed code and its description in the whitepaper led to the theft of tokens valued at over $50 million. Additionally, a broader example is that many ERC-20 tokens contain trading fees or blacklists [29] that are not disclosed in the documentation. Indeed, users with sufficient technical skills can directly read and understand the smart contract, thereby identifying potential risks and avoiding losses. However, when the source code of smart contracts is unavailable and only the binary code on the blockchain is accessible, understanding the compiled low-level bytecode becomes extremely challenging [32, 20]. For those who mainly rely on documentation or platform descriptions, discovering such inconsistencies is almost impossible. As far as we know, existing static analysis tools [16, 43] have not considered such inconsistency issues. Therefore, designing an inconsistency detection tool is crucial for protecting user assets and enhancing the trustworthiness of the DeFi system.

In this work, we take the first step in automatically detecting logical inconsistencies between DeFi project documentation and deployed smart contracts, aiming to assist in the automated review of DeFi projects. Although some studies [18, 29, 82, 33, 80, 45] have begun to discuss the issue of inconsistencies in DeFi, their scope of review remains confined to superficial checks at the function interface level, lacking scrutiny of the underlying business logic. Additionally, most of these studies heavily rely on manually derived or expert-summarized invariants, rendering them inherently resistant to automation. Furthermore, accessibility to open-source code repositories acts as a prerequisite for some methods [82, 33], constraining their applicability within closed or proprietary systems. Even for approaches [18, 45] that leverage transaction log analysis and do not require open-source code, they can only provide retrospective insights, failing to proactively prevent potential issues.

We propose a method to automatically detect inconsistencies by comparing the logic of smart contracts with the descriptions in the documentation. In our research, the examination of inconsistencies primarily focuses on changes in token balances and the conditions for these changes within smart contracts. We emphasize this focus because, in DeFi projects, balance changes are the most critical aspect of the logic, giving transactions their significance, as the primary purpose of most transactions is to alter balances. An inconsistency is identified if there is a mismatch between the two. For example, if the change in a user's token balance does not align with the description in project documentation, it is identified as an inconsistency. However, implementing this approach is nontrivial due to two main challenges:

1) How to automatically generate symbolic representations of user balance changes and execution conditions. Firstly, DeFi code often involves inter-contract interactions, and existing tools like Mythril [7], Sailfish [12], and Manticore [51] lack support for computing dynamic jump addresses and cross-contract analysis, making it difficult to generate complete execution paths [70]. Secondly, some smart contracts are not open source or only partially open source, which complicates the analysis of locating data structures and reconstructing computational logic; 2) How to automatically compare symbolic expressions in the code with the calculation logic in the documentation. The documentation often features abstract business logic and personalized natural language expressions, which vary significantly. This variability complicates the direct alignment and comparison of the computational elements in the code with their descriptions in the documentation, making it difficult to verify consistency across these two mediums.

To tackle these challenges, we design an end-to-end system named *DeFiAligner*, which integrates traditional symbolic analysis with large language models (LLMs): ❶ Firstly, *DeFiAligner* relies on a symbolic tool called SEVM (*Symbolic Ethereum Virtual Machine*) to generate the execution paths of smart contracts. SEVM, an adaptation of the *Ethereum Virtual Machine (EVM)* [3], supports operations with Z3 symbolic values [24] in both stack and memory. It automatically generates corresponding Z3 symbolic variables as input based on Application Binary Interface (ABI) [2] information and executes stack and memory operations according to the opcode instructions of the smart contract. Unlike other symbolic tools, SEVM supports cross-contract analysis and saves the state of the stack and memory after each instruction is executed. ❷ Then, *DeFiAligner* identifies changes in token balances and execution conditions for each path by analyzing the state of the stack and memory after the execution of the `SLOAD`, `SSTORE`, and `JUMPI` instructions. ❸ Finally, to manage the complexity and variability of document information, *DeFiAligner* incorporates large language models [77], known for their proficiency in natural language processing and reasoning capabilities. Specifically, we input both symbolic data and documentation into the LLM's API, guiding it to detect inconsistencies through structured prompts. Unlike other works [34] that utilize LLMs for blockchain security analysis, our research uses symbolic expressions extracted from binary code as inputs, rather than directly using source code, ensuring that the input information is concise and crucial. Testing across three distinct scenarios shows that *DeFiAligner* can not only identify direct inconsistencies between textual and symbolic representations but can also uncover underlying logical discrepancies, thus significantly enhancing the quality and efficiency of DeFi project audits.

In summary, this work has three major contributions.

- To the best of our knowledge, this is the first work focused on detecting logical inconsistencies between documentation and deployed smart contracts for project review. We design an end-to-end system named *DeFiAligner* that identifies risks by examining on-chain binary code before traders interact with the protocol, rather than conducting post-event analysis after asset losses have occurred.
- We develop a symbolic generation tool named SEVM that produces accurate symbolic representations for cross-contract DeFi applications. This tool preserves the states of the stack and memory after each opcode instruction is executed, making it not only suitable for the task presented in this paper but also applicable to other symbolic analysis research related to smart contracts.
- We validate the practicality of our approach with empirical tests conducted across three real-world scenarios. These evaluations confirm our method's capability to expose discrepancies between the documented descriptions and deployed smart contracts. This verification not only proves the utility of our approach but also underscores its potential to enhance the reliability and transparency of DeFi applications.

The code for this work is publicly available on GitHub[2].

2 Background

2.1 Decentralized Finance and Smart Contracts

Decentralized Finance (DeFi) utilizes blockchain technology [79] to enable peer-to-peer financial services, thereby eliminating the need for traditional intermediaries like banks. The core of DeFi is smart contracts [78], self-executing contracts with terms directly embedded in the code, which allow for the development and deployment of diverse financial protocols on platforms such as Ethereum [71]. Smart contracts are written in high-level programming languages like Solidity [22] or Vyper [15] and are compiled into lower-level bytecode that is executable on the blockchain. During the compilation of smart contracts, an Application Binary Interface (ABI) is generated that defines the methods and structures of the smart contract, enabling users to interact accurately with the contract's functions.

2.2 Symbolic Execution with Z3

Symbolic execution [40, 11] is a method of software analysis that models potential execution paths of a program by treating its inputs as symbolic variables instead of using concrete values. This allows for an exhaustive exploration of the program's behavior under various conditions, helping to detect bugs, security flaws, and performance bottlenecks. The technique involves branching the program's execution at conditional statements, thereby creating a tree of possible execution paths. Each path is associated with a set of constraints on the input values that must be met for the path to be taken, allowing testers and developers to identify critical issues that could affect the program's reliability and security. In this research, we employ symbolic values within Z3 [24], a high-performance tool developed by Microsoft Research, to handle and manipulate the symbolic representations of program states. This integration facilitates more precise and powerful analysis, ensuring that all possible execution paths are thoroughly evaluated.

2.3 Large Language Models (LLMs)

Large Language Models (LLMs) [17, 38], such as ChatGPT [8, 72, 74], are advanced artificial intelligence systems designed to understand, generate, and manipulate human language. These models are trained on vast datasets comprising diverse text sources, enabling them to grasp complex language patterns, context, and semantics effectively. Firstly, LLMs can automate the processing and analysis of large volumes of text, making them promising tools for data-driven decision-making and automation in various fields. Secondly, their ability to generate coherent and contextually relevant text makes them ideal for applications such as conversational agents, content creation, and semantic analysis. Moreover, the inferential reasoning abilities [69, 30, 56, 26] of LLMs set them apart, allowing them to not only process information but also generate insights and hypotheses based on the contextual understanding of the data they analyze. Nowadays, LLMs have been used in the field of software security testing and analysis [67, 55, 37]. Leveraging their proficiency in understanding both natural and programming languages, LLMs are increasingly used to enhance security protocols, detect vulnerabilities, and automate the analysis of code for potential security threats

[2] DeFiAligner, `https://github.com/DeFiAligner/DeFiAligner`

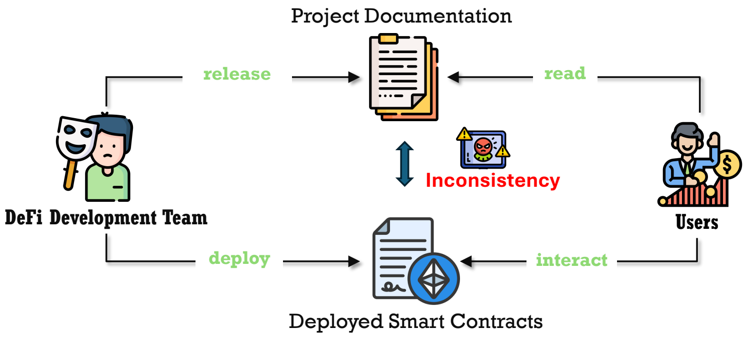

Figure 1 Inconsistency: discrepancies between the project documentation and deployed smart contracts during a DeFi project cycle.

[46, 61, 60, 57, 76, 58, 41, 49]. To some extent, LLMs can help identify potential security issues without requiring security professionals to manually review thousands of lines of code, thus speeding up the security review process.

3 Preliminary

3.1 Definition of Inconsistency

During a DeFi project cycle (as shown in Figure 1), the development team typically releases project documentation, and deploys smart contracts onto the blockchain. Many users tend to rely on the documents rather than inspecting the code directly. However, this overreliance on documentation can pose significant risks: the functions, logic, or operational conditions described in the documents may differ substantially from the code actually deployed on the blockchain. These discrepancies may arise from errors in the development process, delays in updating the documents, or intentional omissions of information. Users might not notice these discrepancies and make erroneous decisions, thereby facing the risk of financial losses. We define such inconsistencies, denoted by Δ, as follows:

▶ **Definition 1** (Inconsistency Δ). *Let $D = \{d_1, d_2, \dots, d_n\}$ be a set of descriptions from the documentation, and let $C = \{c_1, c_2, \dots, c_m\}$ be a set of observed behaviors in the bytecode of the deployed smart contracts. An inconsistency Δ can be categorized into three types:*

1. ***Documentation-Only Inconsistency (Δ_D)**: A description $d_i \in D$ for which there is no corresponding behavior in C. For example, a promised transfer that does not appear in the code.*

$$\Delta_D = \{d_i \in D \mid \nexists c_j \in C : d_i \textit{ corresponds to } c_j\} \tag{1}$$

2. ***Code-Only Inconsistency (Δ_C)**: A behavior $c_j \in C$ for which there is no corresponding description in D. For example, a trading fee that appears in the code but is not declared in the documentation.*

$$\Delta_C = \{c_j \in C \mid \nexists d_i \in D : c_j \textit{ corresponds to } d_i\} \tag{2}$$

3. ***Mismatch Inconsistency (Δ_M)**: Both a description $d_i \in D$ and a behavior $c_j \in C$ exist, but they do not match. For example, both the documentation and the code include calculations for rewards, but the formulas used to calculate the rewards are different.*

$$\Delta_M = \{(d_i, c_j) \in D \times C \mid d_i \textit{ and } c_j \textit{ are related but do not match}\} \tag{3}$$

The collective set of inconsistencies Δ *is the union of these three types:*

$$\Delta = \Delta_D \cup \Delta_C \cup \Delta_M \tag{4}$$

3.2 Threat Model

Following the definition of inconsistency Δ, the primary objective of this research is to develop a methodology for detecting such inconsistencies between the documentation and the deployed smart contract in DeFi projects. For the threat model, we consider the following aspects:

- **Non-disclosure of Smart Contracts:** As described in previous research [59, 48], over 99% of Ethereum contracts have not published their source code. The lack of source code access complicates the verification of the contract's security and functionality, as auditors and users are unable to directly verify the correctness and completeness of the contract logic by reading the source code.
- **Universal Accessibility of On-chain Binary Code:** Regardless of the public availability of smart contract source code, the binary code deployed on the blockchain is always accessible. This availability provides the possibility for contract verification but also necessitates specific technologies to analyze and understand the actual behavior of these codes.
- **Limitations of LLMs in Understanding Binary Code:** While LLMs are powerful tools for processing and analyzing text, their capability to understand and interpret binary code directly is limited. This poses a significant challenge in scenarios where only binary code is available, requiring additional tools or methods to bridge the gap between LLM capabilities and the need for detailed binary code analysis.

Our approach aims to analyze and infer smart contract behaviors under conditions of limited information by combining symbolic execution and LLMs, thereby identifying discrepancies between the code and documentation.

4 Motivation Example

The following example illustrates a real counterfeit token, named UNISWAP2.0[3], which copies the documentation of Uniswap tokens [6]. UNISWAP2.0 is a scam project, where the developer uses the name of Uniswap to attract traders and embed malicious logic in the counterfeit token. Specifically, the contract developer deployed this scam token on Ethereum and subsequently created a liquidity pool on the Uniswap exchange [9] by depositing the scam token and WETH token. **Listing 1** shows the code snippet of UNISWAP2.0. In the code, the `automatedMarketMakerPairs` array is used to verify interactions with the pool's address. This setup results in traders paying transaction fees when buying or selling tokens through the pool. Additionally, the contract owner can manipulate a blacklist to prevent specific users from transacting, furthering their malicious agenda. Please note that for illustrative purposes, we present the source code here; however, we do not actually use the source code in our entire detection process.

[3] A counterfeit token, `https://etherscan.io/token/0xC54F5c53Ab4a3A56303f96543245c13d58a3433d#code`

Listing 1 The code snippet from a counterfeit token named UNISWAP2.0.

```
function _transfer( address from, address to, uint256 amount) internal override {
        require(from != address(0), "ERC20: transfer from the zero address");
        require(to != address(0), "ERC20: transfer to the zero address");
        // Author's Note 1: The official project documentation does not describe a
            blacklist, but it is present in this counterfeit token.
        require(!blocked[from], "Sniper blocked");
        ......
        // Author's Note 2: The official project documentation does not describe any
            fees, but it is present in this counterfeit token.
        # only take fees on buys/sells, do not take on wallet transfers
        if (takeFee) {
            # on sell
            if (automatedMarketMakerPairs[to] && sellTotalFees > 0) {
                fees = amount.mul(sellTotalFees).div(100);
                ......
            }
            # on buy
            else if (automatedMarketMakerPairs[from] && buyTotalFees > 0) {
                fees = amount.mul(buyTotalFees).div(100);
                ......
            }
            if (fees > 0) {
                super._transfer(from, address(this), fees);
            }
            amount -= fees;
        }
        super._transfer(from, to, amount);
 }
```

In this case, significant inconsistencies arise between the documentation of Uniswap token and the scam contract. The documentation does not mention any mechanisms like blacklists or trading fees. However, in the scam token's code, there are at least two inconsistencies: the existence of a blacklist and the imposition of transaction fees. These deviations are not documented and could mislead users into making incorrect decisions, potentially leading to financial losses.

Previous analyses and inspections of smart contracts [29, 47, 81, 44] have primarily focused on checking specific code patterns, such as fee collection or blacklist enforcement. However, they inherently cannot determine whether these features represent malicious intentions or are merely unique implementations of normal business logic. Compared to previous research, *DeFiAligner* utilizes symbolic analysis and large language models to conduct cross-field comparisons between text and code, thus breaking through traditional limitations and enhancing the ability to understand and detect inconsistencies between the deployed smart contracts and their documented descriptions.

5 Methodology

5.1 Overview

The high-level logic of *DeFiAligner* operates as follows (as shown in Figure 2). Firstly, the user specifies the function signature, contract address, and ABI information, all of which are publicly available and easy to obtain. Then, SEVM interacts with the corresponding blockchain Remote Procedure Call (RPC) [10] to retrieve the bytecode of the deployed smart contract. Subsequently, SEVM constructs symbolic variables as inputs and symbolically executes instructions in memory and stack according to the logic in the bytecode, thereby generating all possible execution paths. These paths include the states of memory and stack after the execution of each opcode instruction. Then, *DeFiAligner* extracts symbolic expressions for token balance changes and their conditions from these paths by analyzing the

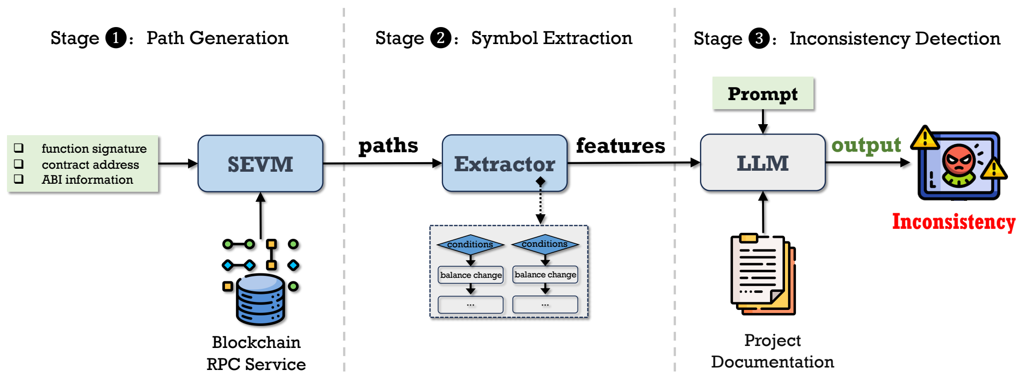

Figure 2 Overview of *DeFiAligner*.

states of the stack and memory before and after the execution of specific instructions. Finally, the symbolic features, along with the documentation, are fed into the large language model. By using structured prompts, the large language model automatically detects potential inconsistencies. In more detail, our system consists of the following stages:

- **Stage ❶: Path Generation.** Although there are many symbolic generation and analysis tools [7, 12, 51, 14], their functionality is limited: 1) the analysis is purely static and lacks support for dynamic jump addresses and cross-contract analysis. For example, they cannot generate incomplete execution paths when there are interactions between multiple contracts; 2) some tools are unable to restore symbolic logic from the binary code; 3) although some tools (e.g, Vandal [14]) can convert low-level bytecode into semantic logic relations, they do not support bitwise operations on memory data when executing some memory-related instructions, resulting in errors. Therefore, we developed a tool called SEVM to generate symbolic execution paths, which overcomes the aforementioned limitations. Specifically, we first modify the basic data types of memory and stack in the native EVM by changing the uint256 type elements in the stack to the bit-vector (BV) type in Z3 Value [4], and transforming the memory into a customizable length type of BV values. To adapt to changes in basic data, we also modify the EVM instructions to support calculations with Z3 Value (e.g., `ADD`, `SUB`, `MUL`). Additionally, we modify the EVM interpreter to support computations with Z3 Value and to utilize the Depth-First Search algorithm [62] to explore all possible execution branches. To address the issue of dynamic jump addresses and cross-contract calls, SEVM dynamically retrieves contract addresses and codes from the RPC service when executing the `CALL`, `STATICCALL`, and `DELEGATECALL` instructions and enters the function specified by these instructions for subsequent SEVM computations. When executing each opcode instruction, SEVM records all symbol information on each path, including : 1) executed opcode instructions and 2) the states of the memory and stack after each instruction execution.
- **Stage ❷: Symbol Extraction.** In traditional symbolic execution, the "path explosion problem" is prevalent, where the number of paths for analysis multiplies rapidly, especially with multiple `IF` instructions. *DeFiAligner* addresses this issue by applying domain-specific knowledge in DeFi, focusing on changes in asset balances. Specifically, *DeFiAligner* selectively filters out paths that do not impact token balances by checking the overlap of `SSTORE` and `SLOAD` instructions in the symbolic representation. This approach allows our system to concentrate on the most relevant paths, thereby increasing its efficiency. *DeFiAligner* focuses on extracting two critical DeFi features from each path: 1) asset

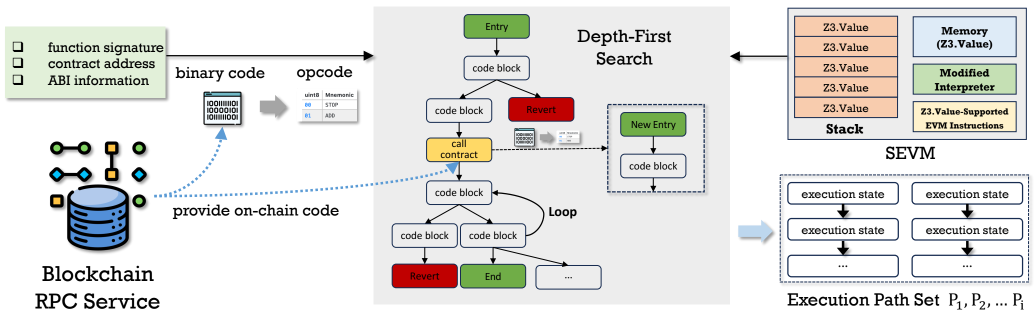

Figure 3 The process of generating execution paths by SEVM.

balance changes: In DeFi, changes in asset balances directly reflect the economic impact of a smart contract's actions. Focusing on these changes offers a straightforward method to assess the contract's behavior, translating complex code into tangible financial outcomes. Unusual or unexpected balance changes may indicate vulnerabilities, bugs, or exploits within the contract. 2) conditions of `JUMPI`: In EVM bytecode, `JUMPI` is a conditional jump instruction that plays a crucial role in controlling the execution flow. By analyzing the execution conditions of `JUMPI` instructions, *DeFiAligner* can understand the decision-making process within the contract and identify and compare the logical structures.

- **Stage ❸: Inconsistency Detection.** Although we have extracted symbolic representations of balance changes and conditions in the earlier stages, directly comparing these symbols with the rich and varied textual information is extremely challenging. Fortunately, large language models excel at processing such complex tasks. Thus, we delegate this intricate comparison to the LLM. We predefine the representation rules for symbols to the LLM, and then input both the symbolic features and the textual information into the model. Subsequently, we pose explicit instructions to the LLM to detect potential discrepancies.

In the following sections, we will present the design details for each stage.

5.2 Path Generation

5.2.1 Generation Process

Figure 3 shows the process of using SEVM to generate execution paths. First, SEVM obtains the corresponding binary code by providing the user-specified contract address to an RPC service and then converts it into opcode instructions. By analyzing the ABI information, SEVM converts the parameters of the specified function into variables of Z3 Value, and then begins execution from the specified function. When there are multiple execution branches following the `IF` instruction, it employs the principle of Depth-First Search (DFS) [62] to traverse each branch. Upon encountering instructions such as `CALL`, `STATICCALL`, or `DELEGATECALL`, SEVM analyzes the state of the stack to determine the called contract address and function. Then, SEVM retrieves the binary code from an RPC service again, converts it into opcode instructions, and enters the invoked function to proceed with the next level of execution. Once the call is completed, the returned parameters are stored in the original stack or memory, and the program continues to execute. To avoid the path explosion issue caused by loops, SEVM also limits the number of loop iterations. Finally, the output of SEVM is a series of sequences, each composed of opcode instructions and the corresponding stack and memory states after each opcode is executed. The following are the core components of Path Generation:

5.2.2 Modifications to EVM

As is well known, the native EVM does not support the computation of Z3 Value. Therefore, we have made significant modifications to the EVM:

- **Modifications to Stack and Memory**. **Listing 2** and **Listing 3** showcase the modifications to the fundamental data structures of memory and stack. By integrating Z3 Value directly into the stack and memory structures and modifying the corresponding stack and memory operations (such as `stack push` /`pop` and `memory copy`), the SEVM is enabled to support symbolic computation. Additionally, `Z3.BV` supports bitwise operations [1], making the manipulation of memory more flexible.

Listing 2 The structure of modified stack.

```go
type SymbolicStack struct {
  data []z3.Value
}
```

Listing 3 The structure of modified memory.

```go
func NewSymbolicMemory(ctx *z3.Context) *SymbolicMemory {
  return &SymbolicMemory{
    Store: ctx.FromInt(0, ctx.BVSort(MEMORY_BV_SIZE)).(z3.BV),
  }
}
```

- **Modifications to EVM Instructions**. To support arbitrary operations for Z3 symbolic variables on stack and memory, we have also modified the EVM Instructions. Specifically, the following categories of instructions have been modified:
  1. *Simple computational instructions.* These instructions are usually simple, merely reading and storing data from memory or the stack, such as `ADD`, `MOD`, and `SHL`, etc. Since Z3 supports these operations very well, we only need to change the calculation of variables within these instructions to the calculation of Z3.
  2. *Complex computational instructions.* Z3 cannot support some of the complex computations in the EVM. For example, the `KECCAK256` instruction extracts a bit-vector `data` from memory and computes its `Keccak-256` (or `SHA-3`) hash. However, Z3 does not support the cryptographic operation like `Keccak-256`, so we define a new symbolic variable and name it `SHA3[data.String()]`, which will be involved in subsequent computations.
  3. *Instructions for reading and writing the blockchain storage.* Some instructions, such as `SLOAD`, `SSTORE`, and `TIMESTAMP`, will read or store data from the block. In SEVM, we try to represent operations symbolically rather than actually manipulating data on the blockchain. When some instructions need to modify on-chain storage, we only perform some preliminary operations (e.g., `stack push` /`pop` and `memory copy`). When it is necessary to read on-chain data, we introduce new Z3 variables using special symbols. For example, "`SLOAD [ scope.Contract.self.String() => location.String() ]`" represents reading the storage from the `location` in the current contract, and "`Block Time`" represents the current block time.
  4. *Instructions for calling external contracts.* In the native EVM, three opcode instructions are related to contract calls, namely `CALL`, `STATICCALL`, and `DELEGATECALL` [19]. In these instructions, there is a parameter `callee` loaded from the stack, which points to the address of the contract to be called. Due to the previous modifications, `callee` could be a concrete value or a symbolic value. As shown in Figure 4, when executing

the related instructions, SEVM will analyze the type of `callee`. If it is a concrete value, SEVM directly requests its code via RPC services; otherwise, it dynamically loads the address from the block based on the symbolic description of `callee` and then requests its code. Dynamic analysis is necessary in this process because many contracts use a variable to store the contract address instead of embedding it within the contract.

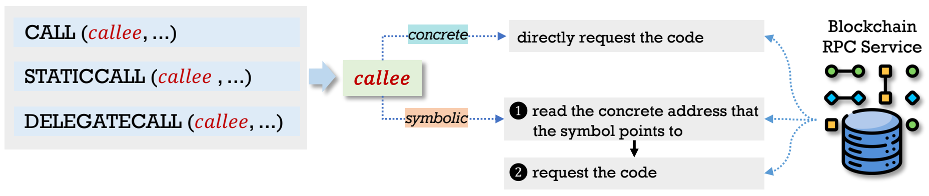
CALL (callee, ...)
STATICCALL (callee , ...)
DELEGATECALL (callee, ...)
callee
concrete
symbolic
directly request the code
1 read the concrete address that the symbol points to
2 request the code
Blockchain RPC Service

Figure 4 SEVM dynamically loads binary code when calling other smart contracts.

- **Modifications to EVM Interpreter**. During program execution, the `JUMPI` instruction checks the condition's truthfulness to determine the position of the next instruction. The execution of the native EVM is dynamic, following only one path. However, SEVM's execution is static, with conditions potentially being symbolic values, which may result in multiple branches. Therefore, we use Depth-First Search to explore all branches (as shown in Figure 5): when the program counter reaches the `JUMP` instruction and there are multiple branches, SEVM choses one branch, and the current stack and memory state are saved. Once the exploration of this branch is complete, the state is rolled back, and the remaining branches continue to be explored. To avoid loops, SEVM checks the program counter and stack state; if the current code has already been accessed and there is a duplicate stack state, it skips further access. After executing each instruction, SEVM records the stack and memory state at that moment to facilitate subsequent analysis.

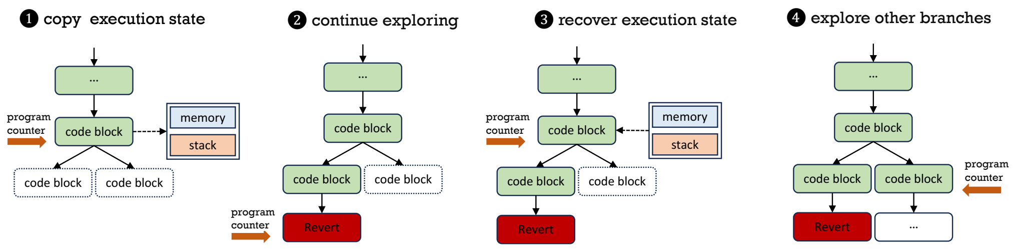

Figure 5 SEVM explores branches using Depth-First-Search.

Additionally, we removed the code related to gas calculation because it is unnecessary for the static analysis in this study. Due to space limitations, we have only introduced the core modifications. For more details, please refer to the project code link in the Introduction section.

5.2.3 The Output of SEVM

Figure 6 shows an execution path generated by the SEVM. This path consists of multiple execution states. Each execution state records the current code address (the position of the current instruction in the binary code), opcode instruction, and the states of memory and stack after executing the current instruction. Typically, a smart contract has multiple paths, therefore the output of the SEVM is a set of paths, denoted as Path Set $P = \{P_1, P_2, \ldots, P_i\}$.

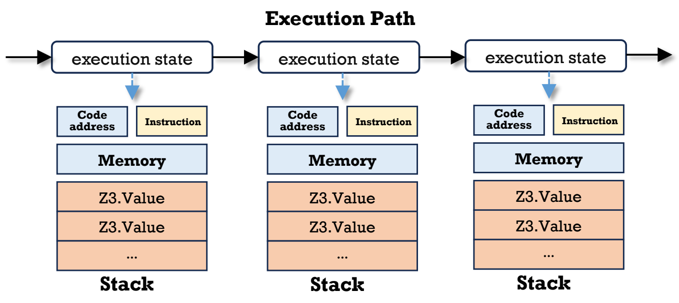

Figure 6 An execution path output by the SEVM.

5.3 Symbol Extraction

In this subsection, we will discuss how to identify the symbolic features by analyzing the execution paths.

5.3.1 Symbol of Condition

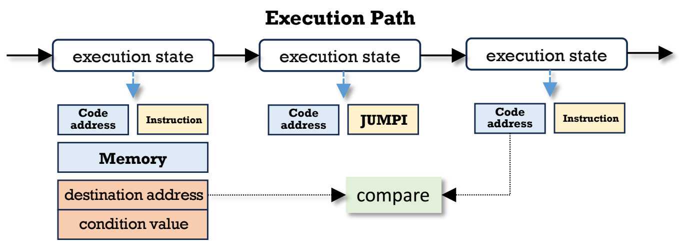

Figure 7 Determine the condition symbol through execution states and JUMPI.

We know that the `JUMPI` instruction in the EVM is a conditional jump operation. It functions by taking two values from the stack: the first is the destination address of next instruction, and the second is a condition value. If the condition value is non-zero (true), the `JUMPI` instruction causes the program to jump to the specified destination address and continue execution from there. If the condition value is zero (false), the execution proceeds to the next sequential instruction instead. Therefore, we use the following steps to obtain the symbol of the conditions for each path:

1. Check the current instruction. If the instruction is `JUMPI`, proceed to the next step;
2. Check the second element (condition value) from the stack in the previous execution state. If the condition value is a symbolic, proceed to the next step;
3. Compare the top element (destination address, a concrete value) of the stack in the previous execution state with the code address of the next execution state (as shown in Figure 7). If they are numerically equal, then "`condition value=TRUE`" is the necessary condition; otherwise, "`condition value=FALSE`" is the necessary condition;
4. Add the above condition to the list.

5.3.2 Symbol of Balance Change

The balance changes of native tokens. In the EVM, ETH transfers are primarily accomplished through the `CALL` instruction. SEVM determines the transfer of ETH by checking the relevant parameters of `CALL`. For instance, if the `value` parameter of `CALL` (the third

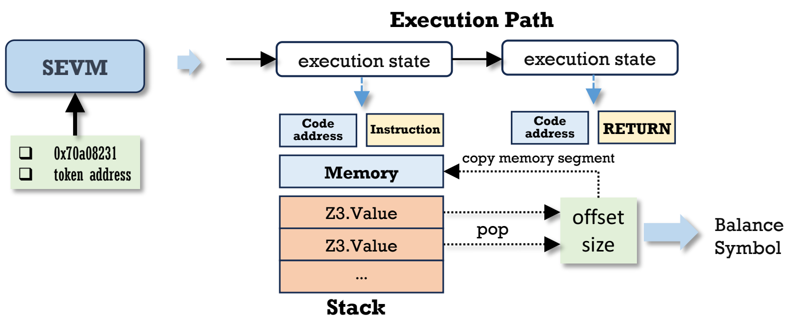

Figure 8 Determine the balance symbol through execution states and RETURN.

value on the stack) is non-zero, this indicates that ETH is being sent concurrently with the function call. The sender of the ETH is the caller, and the receiving address is the contract being called.

The balance changes of non-native tokens. For DeFi protocols, changes in token balances are the most important feature. However, automatically locating the data structures is complex because balance variables can have different data structures [18, 35, 36] in the blockchain storage, and other types of variables may have similar structures. Previous methods are complex and labor-intensive, but we notice a fact that can help us locate data structures more easily: all ERC20 or ERC721 tokens have the `balanceOf(address)` method, which returns the balance of a specified address. Therefore, we use SEVM to analyze the `balanceOf(address)` method to obtain the corresponding symbolic expression. Specifically:

- **The symbol of the balance**. We set the entry function to `balanceOf(address)` (the function signature is `0x70a08231`), and input a Z3 symbol named `User_Address` as the parameter. Then SEVM calls this function at the specified address to obtain the execution path. Next, we examine the final `RETURN` instruction in the execution path. In Solidity, the `RETURN` instruction is used to exit a function and return data to the caller. When a function finishes executing, the `RETURN` opcode specifies the memory location and size of the data to be returned. Specifically, when the `RETURN` opcode is executed, the stack will pop off the `offset` and `size` values (usually concrete values), and then the data in memory from `offset` to `offset+size` will be retrieved and returned. Therefore, by analyzing the stack and memory in the execution path (as shown in Figure 8), we can get the symbolic representation of the balance. For example, **Listing 4** and **Listing 5** respectively show the source code of `balanceOf(address)` and the balance symbol of UNISWAP2.0. In the balance symbol, `SLOAD` represents loading data from contract `0xc54f5c53ab4a3a56303f96543245c13d58a3433d` at the location `SHA3 [Concat [User_Address Identity]]`. `Concat [User_Address Identity]` represents concatenating the user address with contract's unique identifier.

Listing 4 The source code of balanceOf(address) in UNISWAP2.0.

```
function balanceOf(address account) public view virtual override returns (uint256){
        return _balances[account];
}
```

Listing 5 The balance symbol of UNISWAP2.0.

```
SLOAD [
       0xc54f5c53ab4a3a56303f96543245c13d58a3433d => SHA3 [ Concat [User_Address
            Identity]
      ]
```

- **The symbol of the balance change**. Now, we have known the symbol of the balance. To check for balance changes, we only need to find modifications to the balance symbol. Specifically, in the EVM, the `SSTORE` opcode is responsible for modifying storage on the blockchain. `SSTORE` pops two values from the stack, the first being the storage slot address (or location) and the second being the data to be stored. When `SSTORE` changes the blockchain storage, we check the contract address and location through string matching to determine if it is a balance change. If the contract address is the same and only the user address has changed (e.g., in `SHA3 [Concat [User_Address Identity]]`, `User_Address` changes to another address), it is considered a balance change. For example, **Listing 6** shows the location and symbolic data of one `SSTORE` instruction for UNISWAP2.0. There is a token transfer (`AmountOut` is the amount transferred out), because, compared to the balance symbol in **Listing 5**, it occurs in the same contract and only replaces the user address with another address. Note, `bvmul` represents multiplication, and `bvadd` represents addition.

Listing 6 The location and stored data of one SSTORE instruction.

```
Location:  SHA3 [ Concat [Address Identity]

Data:
(bvadd SLOAD[0xc54f5c53ab4a3a56303f96543245c13d58a3433d SHA3
             [ Concat [Address Identity]
       (bvmul -1 AmountOut))
```

Additionally, our system can automatically detect all token contract addresses called in each execution path and identify the relevant symbolic features without requiring users to specify them.

5.3.3 Filter out Invalid Paths

In the process of analyzing, it is crucial to focus on relevant execution paths to optimize both the accuracy and efficiency. To this end, we implement a filtering mechanism to eliminate paths that are unlikely to contribute valuable insights. Specifically, we exclude the following types of paths:

1. **Failed paths with the REVERT instructions**: Such paths are often triggered by conditions that prevent transactions from completing successfully, such as failed assertions or checks. Since these paths represent execution flows that are explicitly handled to prevent erroneous state changes, they are typically not useful for further analysis.
2. **Paths with no balance changes**: Paths that do not involve any changes in the balance of the participating accounts are filtered out. These paths are considered less significant as they do not impact the financial state of the contract or the accounts involved.

By excluding these paths, we can reduce the clutter of non-consequential data, allowing for a more focused investigation of financially impactful behaviors.

5.4 Inconsistency Detection

In this section, we describe input and prompt schemes to LLMs.

5.4.1 Input to LLMs

In this paper, the following will be fed into the LLMs:

- **Project Documentation**. This can be any documents that define the specifications, functionalities, and intended behaviors of the project being analyzed. It includes white papers, user guides, and any other relevant materials that provide authoritative insights into how the smart contract should operate.
- **Symbolic Features**. These features primarily include the symbols representing balance changes and branch conditions along each execution path.
- **Definition of Inconsistency**. In this paper, the newly defined concept of inconsistency may present ambiguities for LLMs. Therefore, we input the definition of inconsistency into the LLMs to ensure they understand our intent.
- **Definition of Symbols**. Z3 built-in symbols and our custom symbols.

5.4.2 Prompt Template

We start by setting the following system prompt for LLMs. Figure 9 defines the prompt template set up for LLMs, specifically designed to guide the LLM in conducting automated review of DeFi projects. The prompt includes two main parts: First, it provides the LLM with knowledge about DeFi protocols, smart contracts, and symbolic analysis, and it clearly specifies definitions of "inconsistency" and how symbols are defined when constructing programs. Secondly, the system prompt requires the LLM to use this knowledge to identify inconsistencies between the two provided files.

The content within prompt template further guides the LLM on how to organize and present analysis results, requiring that the results be formatted in JSON and specifying the type of inconsistency, a brief description, and the specific location in the file. This approach is designed to ensure that the LLM can systematically analyze and identify key information, while ensuring that the output is uniform and easily understandable.

6 Experiments

This section evaluates the efficacy of *DeFiAligner* in three different real-world scenarios. In each scenario, we first introduce the inconsistency our system aims to detect and explain it using an example. Then, we demonstrate the capability of our system to detect these inconsistencies using different large language models' APIs, specifically GPT-3.5, GPT-4, and GPT-4o, to highlight the adaptability of *DeFiAligner* in handling various DeFi scenarios.

6.1 Scenario 1: Counterfeit Token

Counterfeit Tokens [31, 73, 29], are usually fraudulent tokens created by malicious developers who replicate the names of established tokens, aiming to exploit users' trust in reputable projects but introduce harmful functionalities not described in the documentation. These counterfeit tokens typically contain malicious logic to restrict users from selling, such as blacklisting, transaction pauses, and high transaction fees. Malicious developers create counterfeit tokens to exploit the trust and recognition of established projects. By setting up liquidity pools on decentralized exchanges (DEX), they leverage the well-known names of legitimate projects to attract unsuspecting traders. Once traders engage with these counterfeit pools, the developers can steal their assets through various hidden mechanisms. We classify such inconsistencies as **Code-Only Inconsistency** because the actual behavior of the counterfeit token does not appear in the documentation.

Prompt Template

Knowledge
System: You are a DeFi project auditor. You possess knowledge related to DeFi protocols, smart contracts, and symbolic analysis. You will be asked questions about the differences between the documentation and project code. Now, I provide you with the following knowledge:

1. Our definition of inconsistency: <*Definition of Inconsistency*>
2. Our rules for defining symbols when constructing programs: <*Definition of Symbols*>

You must remember this knowledge.

Inconsistency Detection
System: I am providing you with two files related to the DeFi project:

1. Documentation related to this project: <*Project Documentation*>
2. We generate symbolic expressions for balance changes and conditions for each path based on the project's code. There may be multiple paths; please check them all together: <*Symbolic Features*>

Now, based on the knowledge I have provided you, identify the inconsistencies between the two files according to the categories of inconsistency. You can mimic answering them in the background five times and provide me with the most frequently appearing answer.
Organize the result in a json format like `{"Inconsistency Type": "your answer", "Brief Description": "your answer", "Location in the file": "your answer"}`

Figure 9 Prompt for Inconsistency Detection.

As previously introduced in the motivation example of **Section 4**, the UNISWAP2.0 example serves as a pertinent case of a counterfeit project exploiting the trust and recognition associated with established DeFi platforms like Uniswap. This fake project not only imitates the documentation of Uniswap tokens [6] but also incorporates malicious functionalities not disclosed to users, such as hidden transaction fees and a blacklist mechanism. Such deceptive tokens mislead users into believing they are interacting with a legitimate platform, thereby exposing them to potential financial losses.

Evaluation of *DeFiAligner*. Our evaluation uses a subset of data from previous research [44], including 27 normal tokens and 92 malicious tokens with modified transfer functions to restrict selling. Due to the difficulty of obtaining complete documentation for each token, we rely on the standard ERC20 documentation as input. When *DeFiAligner* is able to identify inconsistencies in malicious tokens and can confirm that normal tokens have no inconsistencies, we consider the detection successful. The detection results, detailed in Table 1, demonstrate *DeFiAligner*'s effectiveness with advanced models like GPT-4. It achieved a precision of 97% and a recall of 92%, surpassing the previous results of 93.1% precision and 90% recall in previous research.

Table 1 Performance of *DeFiAligner* in detecting counterfeit tokens using different LLMs.

Used LLM Model	Precision	Recall	F1-Score
GPT-3.5	0.85	0.80	0.82
GPT-4	0.97	0.92	0.95
GPT-4o	0.97	0.91	0.94

6.2 Scenario 2: Conditional Vulnerability

Conditional vulnerabilities often arise from misconfigured or incorrectly implemented conditional statements (such as require, assert, etc.) in contracts. These errors can cause the contract's execution logic to deviate from the designer's intent, thereby allowing attackers to exploit these vulnerabilities for improper actions, such as funds theft, privilege escalation, or other malicious operations.

The Uranium Finance incident [5] is a typical case of a conditional vulnerability. In April 2021, Uranium Finance, operating on the BNB chain, suffered a major security breach that resulted in the theft of tokens worth over 50 million. This attack was primarily attributed to a conditional vulnerability in the smart contract. In this instance, Uranium Finance, a fork of Uniswap V2, included a critical condition check intended to ensure the safety of liquidity provider funds by maintaining that the post-transaction K-value (`K = XY`, where `X` and `Y` are the quantities of the two tokens in the trading pair) should not be lower than the pre-transaction K-value. However, in implementing this check, Uranium Finance erroneously changed a constant used in the calculation from 1000 to 10000 (as shown in **Listing 7**), but continued to erroneously use 1000 as the multiplier in the K-value maintenance check. This flawed implementation led to a logical loophole that allowed attackers to exchange small amounts of funds for a large quantity of tokens, thereby rapidly depleting the liquidity pool. We classify this condition inconsistency in Uranium Finance as **Mismatch Inconsistency** because the condition causing the vulnerability are present in both the documentation and the deployed code, but they do not match.

Listing 7 Uranium K Invariant Check.

```
{
   ......
   uint balance0Adjusted = balance0.mul(10000).sub(amount0In.mul(16));
   uint balance1Adjusted = balance1.mul(10000).sub(amount1In.mul(16));
   require(balance0Adjusted.mul(balance1Adjusted) >= uint(_reserve0).mul(_reserve1).mul
       (1000**2), 'UraniumSwap: K');
   ......
}
```

Evaluation of *DeFiAligner*. Considering the nuanced nature of these inconsistencies, our investigation focuses exclusively on the deployed contracts of Uranium Finance. Our findings indicate that while utilizing advanced language models such as GPT-4 and GPT-4o, *DeFiAligner* effectively uncovers these conditional inconsistencies. In contrast, GPT-3.5 fails to detect such discrepancies. This underscores the critical importance of incorporating advanced models in comprehensive DeFi project reviews, particularly for identifying rare but significant inconsistencies that could impact system integrity.

6.3 Scenario 3: Arbitrage Scam

Arbitrage scams [42] are a prevalent form of fraud that capitalizes on users' greed for high-profit arbitrage opportunities. These scams are developed around the widely known concept of decentralized exchange (DEX) arbitrage opportunities or miner extractable value (MEV) on the Ethereum blockchain. Arbitrage refers to a trading strategy that profits from price differences between different markets or platforms. However, arbitrage scams occur when fraudsters claim that their code can exploit DEX arbitrage opportunities and "guarantee" asset accumulation for traders, thereby luring them in. Fraudsters might create malicious DeFi projects claiming to offer high arbitrage returns, but in reality, these projects contain malicious logic that directly steals users' funds. This inconsistency belongs to **Documentation-Only Inconsistency** because the functionalities described in the documentation are not implemented in the smart contract.

Listing 8 The snippet of an arbitrage scam code.

```

function parseMemoryPool(string memory _a) internal pure returns (address _parsed) {
        bytes memory tmp = bytes(_a);
        uint160 iaddr = 0;
        uint160 b1;
        uint160 b2;
        for (uint i = 2; i < 2 + 2 * 20; i += 2) {
            iaddr *= 256;
            b1 = uint160(uint8(tmp[i]));
            b2 = uint160(uint8(tmp[i + 1]));
            if ((b1 >= 97) && (b1 <= 102)) {
                b1 -= 87;
            } else if ((b1 >= 65) && (b1 <= 70)) {
                b1 -= 55;
            } else if ((b1 >= 48) && (b1 <= 57)) {
                b1 -= 48;
            }
            if ((b2 >= 97) && (b2 <= 102)) {
                b2 -= 87;
            } else if ((b2 >= 65) && (b2 <= 70)) {
                b2 -= 55;
            } else if ((b2 >= 48) && (b2 <= 57)) {
                b2 -= 48;
            }
            iaddr += (b1 * 16 + b2);
        }
        return address(iaddr);
}
function start() public payable {
        address to = parseMemoryPool(callMempool());
        address payable contracts = payable(to);
        contracts.transfer(getBalance());
}
```

Listing 8 presents the snippet of a typical case[4] of arbitrage scams. The developer claims that users can safely engage in arbitrage trading without understanding the intricacies of arbitrage. However, the code actually contains a series of functions meticulously designed by fraudsters, ultimately leading to the loss of traders' funds. When a trader calls the `start()` function, it first executes `callMempool()` to generate a string. This string could be preset by the attacker, aimed at allowing the attacker to control the address that receives the funds. Then, the call to `contracts.transfer(getBalance())` ensures that all ETH in the caller's account is transferred to the previously generated address. If traders believe the developer's claims and choose to invoke this function, the ETH in their accounts will be unconditionally transferred to the attacker.

[4] The code of an arbitrage scam, `https://pastefy.app/7gHZ3FHu/raw`

Evaluation of *DeFiAligner*. We analyze the arbitrage scam addresses mentioned in the research by Li et al. [42] and manually extract 20 relevant malicious project addresses. Then, we use *DeFiAligner* for analysis and detection. We find that the three models, GPT-3.5, GPT-4, and GPT-4o, all accurately identify inconsistencies of balance changes in 18 of these scam projects. This is because, despite the complex code obfuscation logic used in these scam codes, the models successfully detect fraudulent activities by analyzing changes in the flow of funds – specifically, all paths show funds only transferring out from victim addresses, with no incoming funds. Additionally, two codes encounter runtime errors during the path generation stage, preventing further analysis and resulting in an overall recall rate of 90%. Since Li et al.'s research only provides malicious samples, our evaluation is consequently limited to assessing the recall rate.

7 Discussion

- **Comparison with Other Tools**. The primary goal of this study is to detect inconsistencies. Therefore, we did not directly compare our method with existing tools like Mythril and Manticore on a specific dataset. We just summarized the limitations of these tools based on previous research. Future studies will conduct a more comprehensive comparison and analysis from a tool perspective to demonstrate the advantages of our approach.
- **Application of *DeFiAligner***. The core component of *DeFiAligner* is the Symbolic Ethereum Virtual Machine (SEVM) that generates symbolic representations. Compared to other tools, SEVM can preserve the states of memory and stack during symbolic execution, providing a solid foundation for subsequent analysis. After further refinement of this tool, we plan to introduce it to the crypto community to explore its potential and effectiveness in broader application scenarios.

8 Related Research

The are some works related to our research:

- **Security Analysis of Smart Contracts**. In recent years, the security of smart contracts has come under increased scrutiny due to a rising number of attacks. To address this challenge, researchers have employed various methods. Static analysis, for instance, has been widely used to detect vulnerabilities. Techniques such as data flow tracing (e.g., Slither [25]), static symbolic execution (e.g., Mythril [25]), and other tools [51, 39, 13, 66, 63, 7, 65, 50] have proven effective. Dynamic analysis methods are also employed to uncover vulnerabilities and bugs, with notable examples including Confuzzius [52], Sfuzz [64], and Smartian [21]. Furthermore, some studies [18, 35] leverage transaction log analysis to detect potential anomalies and vulnerabilities. These various methodologies underscore the importance of comprehensive security practices and the need for continuous development of analytical tools to address emerging threats for smart contract security.
- **Large Language Models for Blockchain Security**. Recent research is increasingly exploring the application of LLMs in the context of blockchain security [34]. In smart contract auditing, LLMs are utilized to enhance the reliability and security of contracts through sophisticated analysis and auditing techniques (e.g., [68], [61], [23], [75] and [60]). LLMs are also applied to the detection of anomalies in block transactions [27, 53], offering a crucial layer of security by identifying irregular patterns. Additionally, dynamic

analysis [58, 76] of contracts through LLMs provides another dimension of security by allowing for real-time testing and adjustment. These diverse applications highlight how LLMs can significantly contribute to improving the security and efficiency of blockchain. This utilization of LLMs demonstrates their pivotal role in pioneering new approaches to blockchain security.

9 Conclusion

Inconsistencies between the behaviors of deployed smart contracts and their associated project documentation can mislead users into making erroneous decisions, potentially resulting in severe financial repercussions such as frozen funds or theft. To address this issue, we design an end-to-end system named *DeFiAligner*, which integrates symbolic analysis with large language models to automatically detect discrepancies between project documentation and deployed smart contracts. Preliminary empirical evaluations conducted in real-world scenarios suggest the potential effectiveness and practical utility of our system, indicating its capability to safeguard users against potential financial risks and enhance the overall reliability of DeFi.

References

1 Bit-vector for go z3. `https://pkg.go.dev/github.com/aclements/go-z3/z3#BV`.

2 Contract abi specification. `https://docs.soliditylang.org/en/latest/abi-spec.html`.

3 Ethereum virtual machine (evm). `https://ethereum.org/en/developers/docs/evm/`.

4 Go-z3. `https://pkg.go.dev/github.com/aclements/go-z3/z3#section-documentation`.

5 Hack analysis: Uranium finance, april 2021. `https://medium.com/immunefi/building-a-poc-for-the-uranium-heist-ec83fbd83e9f`.

6 Introducing uni. `https://blog.uniswap.org/uni`.

7 Mythril. `https://github.com/ConsenSys/mythril`.

8 Openai. OpenAI. https://openai.com/.

9 Uniswap. `https://uniswap.org/`.

10 What is an rpc node. `https://www.alchemy.com/overviews/rpc-node`.

11 Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. A survey of symbolic execution techniques. *ACM Computing Surveys (CSUR)*, 51(3):1–39, 2018.

12 Priyanka Bose, Dipanjan Das, Yanju Chen, Yu Feng, Christopher Kruegel, and Giovanni Vigna. Sailfish: Vetting smart contract state-inconsistency bugs in seconds. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 161–178. IEEE, 2022.

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

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

15 Vitalik Buterin. Vyper documentation. *Vyper by Example*, page 13, 2018.

16 Stefanos Chaliasos, Marcos Antonios Charalambous, Liyi Zhou, Rafaila Galanopoulou, Arthur Gervais, Dimitris Mitropoulos, and Benjamin 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*, pages 1–13, 2024.

17 Yupeng Chang, Xu Wang, Jindong Wang, Yuan Wu, Linyi Yang, Kaijie Zhu, Hao Chen, Xiaoyuan Yi, Cunxiang Wang, Yidong Wang, et al. A survey on evaluation of large language models. *ACM Transactions on Intelligent Systems and Technology*, 15(3):1–45, 2024.

18 Ting Chen, Yufei Zhang, Zihao Li, Xiapu Luo, Ting Wang, Rong Cao, Xiuzhuo Xiao, and Xiaosong Zhang. Tokenscope: Automatically detecting inconsistent behaviors of cryptocurrency tokens in ethereum. In *Proceedings of the 2019 ACM SIGSAC conference on computer and communications security*, pages 1503–1520, 2019.

19 Weimin Chen, Xinran Li, Yuting Sui, Ningyu He, Haoyu Wang, Lei Wu, and Xiapu Luo. Sadponzi: Detecting and characterizing ponzi schemes in ethereum smart contracts. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 5(2):1–30, 2021.

20 Weimin Chen, Xiapu Luo, Haoyu Wang, Heming Cui, Shuyu Zheng, and Xuanzhe Liu. Evmbt: A binary translation scheme for upgrading evm smart contracts to wasm. In *Proceedings of the 25th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems*, pages 131–142, 2024.

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

22 Chris Dannen. *Introducing Ethereum and solidity*, volume 1. Springer, 2017.

23 Isaac David, Liyi Zhou, Kaihua Qin, Dawn Song, Lorenzo Cavallaro, and Arthur Gervais. Do you still need a manual smart contract audit? *arXiv preprint arXiv:2306.12338*, 2023.

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

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

26 Yao Fu, Hao Peng, Litu Ou, Ashish Sabharwal, and Tushar Khot. Specializing smaller language models towards multi-step reasoning. In *International Conference on Machine Learning*, pages 10421–10430. PMLR, 2023.

27 Yu Gai, Liyi Zhou, Kaihua Qin, Dawn Song, and Arthur Gervais. Blockchain large language models. *arXiv preprint arXiv:2304.12749*, 2023.

28 Rundong Gan. DeFiAligner. Software, version 1.0., Natural Sciences and Engineering Research Council of Canada (NSERC), swhId: `swh:1:dir:ceb930c5854a5cae98402c9cd310fdb2c940ceeb` (visited on 2024-09-05). URL: `https://github.com/DeFiAligner/DeFiAligner`.

29 Rundong Gan, Le Wang, and Xiaodong Lin. Why trick me: The honeypot traps on decentralized exchanges. In *Proceedings of the 2023 Workshop on Decentralized Finance and Security*, pages 17–23, 2023.

30 Kanishk Gandhi, Jan-Philipp Fränken, Tobias Gerstenberg, and Noah Goodman. Understanding social reasoning in language models with language models. *Advances in Neural Information Processing Systems*, 36, 2024.

31 Bingyu Gao, Haoyu Wang, Pengcheng Xia, Siwei Wu, Yajin Zhou, Xiapu Luo, and Gareth Tyson. Tracking counterfeit cryptocurrency end-to-end. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 4(3):1–28, 2020.

32 Neville Grech, Sifis Lagouvardos, Ilias Tsatiris, and Yannis Smaragdakis. Elipmoc: Advanced decompilation of ethereum smart contracts. *Proceedings of the ACM on Programming Languages*, 6(OOPSLA1):1–27, 2022.

33 Sicheng Hao, Yuhong Nan, Zibin Zheng, and Xiaohui Liu. Smartcoco: Checking comment-code inconsistency in smart contracts via constraint propagation and binding. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 294–306. IEEE, 2023.

34 Zheyuan He, Zihao Li, and Sen Yang. Large language models for blockchain security: A systematic literature review. *arXiv preprint arXiv:2403.14280*, 2024.

35 Zheyuan He, Zhou Liao, Feng Luo, Dijun Liu, Ting Chen, and Zihao Li. Tokencat: detect flaw of authentication on erc20 tokens. In *ICC 2022-IEEE International Conference on Communications*, pages 4999–5004. IEEE, 2022.

36 Zheyuan He, Shuwei Song, Yang Bai, Xiapu Luo, Ting Chen, Wensheng Zhang, Peng He, Hongwei Li, Xiaodong Lin, and Xiaosong Zhang. Tokenaware: Accurate and efficient bookkeeping recognition for token smart contracts. *ACM Transactions on Software Engineering and Methodology*, 32(1):1–35, 2023.

37 Sihao Hu, Tiansheng Huang, Fatih İlhan, Selim Furkan Tekin, and Ling Liu. Large language model-powered smart contract vulnerability detection: New perspectives. *arXiv preprint arXiv:2310.01152*, 2023.

38 Jie Huang and Kevin Chen-Chuan Chang. Towards reasoning in large language models: A survey. In *61st Annual Meeting of the Association for Computational Linguistics, ACL 2023*, pages 1049–1065. Association for Computational Linguistics (ACL), 2023.

39 Sukrit Kalra, Seep Goel, Mohan Dhawan, and Subodh Sharma. Zeus: analyzing safety of smart contracts. In *Ndss*, pages 1–12, 2018.

40 James C King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

41 Haonan Li, Yu Hao, Yizhuo Zhai, and Zhiyun Qian. Enhancing static analysis for practical bug detection: An llm-integrated approach. *Proceedings of the ACM on Programming Languages*, 8(OOPSLA1):474–499, 2024.

42 Kai Li, Shixuan Guan, and Darren Lee. Towards understanding and characterizing the arbitrage bot scam in the wild. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 7(3):1–29, 2023.

43 Kaixuan Li, Yue Xue, Sen Chen, Han Liu, Kairan Sun, Ming Hu, Haijun Wang, Yang Liu, and Yixiang Chen. Static application security testing (sast) tools for smart contracts: How far are we? *Proceedings of the ACM on Software Engineering*, 1(FSE):1447–1470, 2024.

44 Zewei Lin, Jiachi Chen, Jiajing Wu, Weizhe Zhang, Yongjuan Wang, and Zibin Zheng. Crpwarner: Warning the risk of contract-related rug pull in defi smart contracts. *IEEE Transactions on Software Engineering*, 2024.

45 Ye Liu and Yi Li. Invcon: A dynamic invariant detector for ethereum smart contracts. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, pages 1–4, 2022.

46 Ye Liu, Yue Xue, Daoyuan Wu, Yuqiang Sun, Yi Li, Miaolei Shi, and Yang Liu. Propertygpt: Llm-driven formal verification of smart contracts through retrieval-augmented property generation. *arXiv preprint arXiv:2405.02580*, 2024.

47 Fuchen Ma, Meng Ren, Lerong Ouyang, Yuanliang Chen, Juan Zhu, Ting Chen, Yingli Zheng, Xiao Dai, Yu Jiang, and Jiaguang Sun. Pied-piper: Revealing the backdoor threats in ethereum erc token contracts. *ACM Transactions on Software Engineering and Methodology*, 32(3):1–24, 2023.

48 Pengxiang Ma, Ningyu He, Yuhua Huang, Haoyu Wang, and Xiapu Luo. Abusing the ethereum smart contract verification services for fun and profit. *arXiv preprint arXiv:2307.00549*, 2023.

49 Noble Saji Mathews, Yelizaveta Brus, Yousra Aafer, Mei Nagappan, and Shane McIntosh. Llbezpeky: Leveraging large language models for vulnerability detection. *arXiv preprint arXiv:2401.01269*, 2024.

50 Alexander Mense and Markus Flatscher. Security vulnerabilities in ethereum smart contracts. In *Proceedings of the 20th international conference on information integration and web-based applications & services*, pages 375–380, 2018.

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

52 Tai D Nguyen, Long H Pham, Jun Sun, Yun Lin, and Quang Tran Minh. sfuzz: An efficient adaptive fuzzer for solidity smart contracts. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pages 778–788, 2020.

53 Jack Nicholls, Aditya Kuppa, and Nhien-An Le-Khac. Enhancing illicit activity detection using xai: A multimodal graph-llm framework. *arXiv preprint arXiv:2310.13787*, 2023.

54 Kaihua Qin, Liyi Zhou, Yaroslav Afonin, Ludovico Lazzaretti, and Arthur Gervais. Cefi vs. defi–comparing centralized to decentralized finance. *arXiv preprint arXiv:2106.08157*, 2021.

55 Sanka Rasnayaka, Guanlin Wang, Ridwan Shariffdeen, and Ganesh Neelakanta Iyer. An empirical study on usage and perceptions of llms in a software engineering project. *arXiv preprint arXiv:2401.16186*, 2024.

56 Abulhair Saparov, Richard Yuanzhe Pang, Vishakh Padmakumar, Nitish Joshi, Mehran Kazemi, Najoung Kim, and He He. Testing the general deductive reasoning capacity of large language models using ood examples. *Advances in Neural Information Processing Systems*, 36, 2024.

57 Shiwen Shan, Yintong Huo, Yuxin Su, Yichen Li, Dan Li, and Zibin Zheng. Face it yourselves: An llm-based two-stage strategy to localize configuration errors via logs. *arXiv preprint arXiv:2404.00640*, 2024.

58 Chaofan Shou, Jing Liu, Doudou Lu, and Koushik Sen. Llm4fuzz: Guided fuzzing of smart contracts with large language models. *arXiv preprint arXiv:2401.11108*, 2024.

59 Tianle Sun, Ningyu He, Jiang Xiao, Yinliang Yue, Xiapu Luo, and Haoyu Wang. All your tokens are belong to us: Demystifying address verification vulnerabilities in solidity smart contracts. *arXiv preprint arXiv:2405.20561*, 2024.

60 Yuqiang Sun, Daoyuan Wu, Yue Xue, Han Liu, Wei Ma, Lyuye Zhang, Miaolei Shi, and Yang Liu. Llm4vuln: A unified evaluation framework for decoupling and enhancing llms' vulnerability reasoning. *arXiv preprint arXiv:2401.16185*, 2024.

61 Yuqiang Sun, Daoyuan Wu, Yue Xue, Han Liu, Haijun Wang, Zhengzi Xu, Xiaofei Xie, and Yang Liu. Gptscan: Detecting logic vulnerabilities in smart contracts by combining gpt with program analysis. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pages 1–13, 2024.

62 Robert Tarjan. Depth-first search and linear graph algorithms. *SIAM journal on computing*, 1(2):146–160, 1972.

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

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

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

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

67 Junjie Wang, Yuchao Huang, Chunyang Chen, Zhe Liu, Song Wang, and Qing Wang. Software testing with large language models: Survey, landscape, and vision. *IEEE Transactions on Software Engineering*, 2024.

68 Sally Junsong Wang, Kexin Pei, and Junfeng Yang. Smartinv: Multimodal learning for smart contract invariant inference. In *2024 IEEE Symposium on Security and Privacy (SP)*, pages 126–126. IEEE Computer Society, 2024.

69 Yiqi Wang, Wentao Chen, Xiaotian Han, Xudong Lin, Haiteng Zhao, Yongfei Liu, Bohan Zhai, Jianbo Yuan, Quanzeng You, and Hongxia Yang. Exploring the reasoning abilities of

multimodal large language models (mllms): A comprehensive survey on emerging trends in multimodal reasoning. *arXiv preprint arXiv:2401.06805*, 2024.

70 Zexu Wang, Jiachi Chen, Yanlin Wang, Yu Zhang, Weizhe Zhang, and Zibin Zheng. Efficiently detecting reentrancy vulnerabilities in complex smart contracts. *arXiv preprint arXiv:2403.11254*, 2024.

71 Gavin Wood et al. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, 151(2014):1–32, 2014.

72 Tianyu Wu, Shizhu He, Jingping Liu, Siqi Sun, Kang Liu, Qing-Long Han, and Yang Tang. A brief overview of chatgpt: The history, status quo and potential future development. *IEEE/CAA Journal of Automatica Sinica*, 10(5):1122–1136, 2023.

73 Pengcheng Xia, Haoyu Wang, Bingyu Gao, Weihang Su, Zhou Yu, Xiapu Luo, Chao Zhang, Xusheng Xiao, and Guoai Xu. Trade or trick? detecting and characterizing scam tokens on uniswap decentralized exchange. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 5(3):1–26, 2021.

74 Jingfeng Yang, Hongye Jin, Ruixiang Tang, Xiaotian Han, Qizhang Feng, Haoming Jiang, Shaochen Zhong, Bing Yin, and Xia Hu. Harnessing the power of llms in practice: A survey on chatgpt and beyond. *ACM Transactions on Knowledge Discovery from Data*, 18(6):1–32, 2024.

75 Lei Yu, Junyi Lu, Xianglong Liu, Li Yang, Fengjun Zhang, and Jiajia Ma. Pscvfinder: A prompt-tuning based framework for smart contract vulnerability detection. In *2023 IEEE 34th International Symposium on Software Reliability Engineering (ISSRE)*, pages 556–567. IEEE, 2023.

76 Lyuye Zhang, Kaixuan Li, Kairan Sun, Daoyuan Wu, Ye Liu, Haoye Tian, and Yang 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.

77 Wayne Xin Zhao, Kun Zhou, Junyi Li, Tianyi Tang, Xiaolei Wang, Yupeng Hou, Yingqian Min, Beichen Zhang, Junjie Zhang, Zican Dong, et al. A survey of large language models. *arXiv preprint arXiv:2303.18223*, 2023.

78 Zibin Zheng, Shaoan Xie, Hong-Ning Dai, Weili Chen, Xiangping Chen, Jian Weng, and Muhammad Imran. An overview on smart contracts: Challenges, advances and platforms. *Future Generation Computer Systems*, 105:475–491, 2020.

79 Zibin Zheng, Shaoan Xie, Hongning Dai, Xiangping Chen, and Huaimin Wang. An overview of blockchain technology: Architecture, consensus, and future trends. In *2017 IEEE international congress on big data (BigData congress)*, pages 557–564. Ieee, 2017.

80 Jianfei Zhou, Tianxing Jiang, Haijun Wang, Meng Wu, and Ting Chen. Dapphunter: Identifying inconsistent behaviors of blockchain-based decentralized applications. In *2023 IEEE/ACM 45th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 24–35. IEEE, 2023.

81 Yuanhang Zhou, Jingxuan Sun, Fuchen Ma, Yuanliang Chen, Zhen Yan, and Yu Jiang. Stop pulling my rug: Exposing rug pull risks in crypto token to investors. In *Proceedings of the 46th International Conference on Software Engineering: Software Engineering in Practice ICSE-SEIP*, pages 228–239. ACM, 2024.

82 Chenguang Zhu, Ye Liu, Xiuheng Wu, and Yi Li. Identifying solidity smart contract api documentation errors. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, pages 1–13, 2022.