Guardians of the Ledger: Protecting Decentralized Exchanges From State Derailment Defects

Zongwei Li, Wenkai Li, Xiaoqi Li, *Member, IEEE*, and Yuqing Zhang, *Member, IEEE*

***Abstract*—The decentralized exchange (DEX) leverages smart contracts to trade digital assets for users on the blockchain. Developers usually develop several smart contracts into one project, implementing complex logic functions and multiple transaction operations. However, the interaction among these contracts poses challenges for developers analyzing the state logic. Due to the complex state logic in DEX projects, many critical state derailment defects have emerged in recent years. In this article, we conduct the first systematic study of state derailment defects in DEX. We define five categories of state derailment defects and provide detailed analyses of them. Furthermore, we propose a novel deep learning-based framework StateGuard for detecting state derailment defects in DEX smart contracts. It leverages a smart contract deconstructor to deconstruct the contract into an abstract syntax tree (AST), from which five categories of dependency features are extracted. Next, it implements a graph optimizer to process the structured data. At last, the optimized data is analyzed by graph convolutional networks to identify potential state derailment defects. We evaluated StateGuard through a dataset of 46 DEX projects containing 5671 smart contracts, and it achieved 94.25% F1-score. In addition, in a comparison experiment with state-of-the-art, StateGuard leads the F1-score by 6.29%. To further verify its practicality, we used StateGuard to audit real-world contracts and successfully authenticated multiple novel common vulnerabilities and exposures.**

***Index Terms*—Decentralized exchange (DEX), deep learning, defect, graph convolutional network (GCN), smart contract.**

I. Introduction

THE decentralized exchanges (DEXs) play a crucial role in decentralized finance (DeFi), enabling direct peer-to-peer transactions without intermediaries [1]. Empowered by smart contracts, DEXs facilitate direct interaction between market participants, departing from the traditional reliance on intermediaries like centralized exchanges (CEXs) [2]. In contrast, DEX leverages smart contracts to eliminate the central point of CEX, implementing user-managed assets throughout the transaction process. Thereby, it reduces risks associated with the central exchange being hacked.

Received 30 August 2024; accepted 26 November 2024. This work was supported by the National Natural Science Foundation of China under Grant 62402146 and Grant 62362021. Associate Editor: Y. Li. *(Corresponding author: Xiaoqi Li.)*

Zongwei Li, Wenkai Li, and Xiaoqi Li are with the School of Cyberspace Security, Hainan University, Haikou 570228, China (e-mail: lizw1017@gmail.com; liwenkai871@gmail.com; csxqli@gmail.com).

Yuqing Zhang is with the National Computer Network Intrusion Protection Center, University of Chinese Academy of Sciences, Beijing 100049, China (e-mail: zhangyq@nipc.org.cn).

Digital Object Identifier 10.1109/TR.2024.3509414

With the development of DeFi, DEX suffers from many security issues. DEXs struggle with defects to various attacks, including fund theft, market manipulation, and denial of service [3]. For instance, the FixedFloat exchange was exploited by an access control defect with the third-party infrastructure, leading to around $26 million being stolen in February 2024 [4].

Several previous studies utilized different techniques (e.g., symbolic execution, data invariant detection, chain synchronization) [5], [6], [7], [8], [9] to detect state inconsistency vulnerabilities in DEXs. Geoffrey et al. [1] introduced SPEEDEX to combat front-running attacks and enhance transaction parallelization in CEXs. In a different approach, Duan et al. [10] developed VetSC, a tool for automated security checks in decentralized applications (DApps). Li et al. [11] proposed SolSaviour, a framework for repairing flawed smart contracts. However, despite these advancements, detecting, and mitigating state defects in smart contracts remain challenging.

Challenge 1 (C1). Complex state logic: The blockchain-based DEX project processes many transactions in a distributed manner, and any related transaction can affect state information. However, due to the complex state logic encapsulated within the DEX contracts, state changes manipulated by an attacker can lead to logic errors in an unpredictable manner. Its higher complexity and interoperability challenge understanding and detecting state changes caused by attacks and malicious behaviors. This is because traditional methods have difficulty capturing complex contracts' structure and interactions when dealing with DEX. They have difficulty in accurately understanding the state changes between contracts.

Challenge 2 (C2). State derailment defects: State derailment defects are distinct from other state defects [12] and are a specific category of security defects in DEXs. Fig. 1 represents how these defects can be exploited, resulting in state derailment. These defects originate from various problems, including logical inconsistencies, resource limitations, access control issues, type and declaration errors, and inadequate exception handling.

State derailment and state inconsistency are critical concepts in understanding the integrity and reliability of system states. State derailment refers to aberrant state behavior where the system deviates from its expected functionality due to unauthorized alterations or erroneous state updates. This deviation can lead to significant disruptions, such as unexpected contract behavior or system failures, thereby compromising the system's reliability and predictability. For instance, a state derailment might occur if a financial transaction system erroneously processes a payment

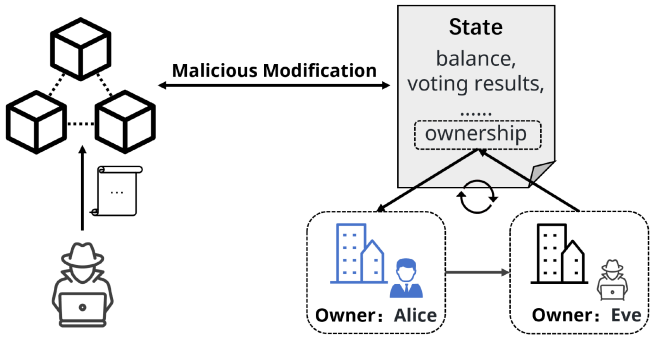

Fig. 1. Illustration of malicious state modification.

due to an unauthorized state change, causing financial discrepancies.

On the other hand, state inconsistency pertains to discrepancies or conflicts between various replicas of the same state at the data level. This issue arises when different users or systems access different versions of the same state, leading to a lack of uniformity and potentially causing confusion or errors in decision-making processes. For example, in a distributed database, state inconsistency might occur if one server shows an account balance of $100 while another shows $150, leading to conflicting information being presented to users. While state derailment focuses on the deviation from expected behavior due to incorrect state changes, state inconsistency highlights the challenges of maintaining uniformity across multiple state replicas.

Our solution: To address these challenges, we propose a deep learning-based framework called StateGuard to analyze complex state logic and detect state derailment defects in DEX smart contracts.

Solution for C1: We propose a smart contract deconstructor that can handle various versions of solidity and AST. It can deconstruct contracts on DEX into ASTs that are easier to analyze. Moreover, it can adaptively extract critical dependency features from the AST, obtaining structural and semantic characteristics of the source code.

Solution for C2: After extracting the dependency features, we integrate node attributes and critical paths to improve the graph representation. This optimized graph was fed into a GCN model to identify and learn defective features to detect state derailment defects. The GCN not only handles the attribute information of the nodes but also considers the connectivity relationships between the nodes. With the advantages of GCN, StateGuard can deeply understand the complex logic and state changes in DEX smart contracts, providing more accurate analysis.

The main contributions of this article are as follows.

1) To the best of authors' knowledge, we conduct the first systematic study of state derailment defects in DEX contracts. We define five kinds of state derailment defects, which can lead to unauthorized or incorrect modifications to the system state during the execution (Section III).
2) We propose StateGuard, a novel deep learning-based framework for detecting and analyzing state defects in DEX projects. It learns structural features from the ASTs of DEX contracts and extracts dependent features to identify state derailment defects (Section IV).
3) We evaluated StateGuard on 46 DEX projects containing 5671 smart contracts with 94.25% F1-score. We also conducted a comparative analysis with state-of-the-arts, with the advantages of 6.29% in F1-score. In addition, StateGuard has discovered multiple novel real-world defects, e.g., common vulnerabilities and exposure (CVE)-2023-{47033, 47034, 47035} (Section V).
4) We open source StateGuard's codes and experimental data at.[1]

II. Background

A. Ethereum and Smart Contract

The rapid digitalization of society necessitates a secure, efficient, and transparent mechanism for data exchange. With its unique, secure structure of chained data blocks, blockchain technology offers a promising solution. However, early blockchain instances like Bitcoin have limited functionality. Ethereum [13], introduced by Vitalik Buterin in 2013, addresses these limitations by broadening the utility of blockchain through smart contracts. These self-executing programs allow secure, trustless transactions without intermediaries. Ethereum's versatility has led to the development of various DApps, such as those in DeFi, providing users with transparent, secure, and fair services.

A diverse range of DApps, including DeFi applications and DEX, can be developed on the Ethereum platform. These applications leverage the capabilities of smart contracts to furnish users with equitable, transparent, and secure services. However, Ethereum confronts several significant challenges. For instance, due to Ethereum's constrained computational capacity, a high volume of transactions can precipitate network congestion, impacting the user experience. Smart contract security is a concern due to coding defects that may lead to financial losses [14].

B. Decentralized Application

DApp is an application that operates on a blockchain network, free from control by any central authority or individual [15]. The emergence of this application model offers users an exceptionally secure, transparent, and efficient service platform, enhancing the security and reliability of data storage and transmission. In DApp, users use the application via mobile devices or other clients. They execute smart contracts, record transactions, and verify them on the blockchain network. As depicted in Fig. 2, this process guarantees the decentralization, immutability, and transparency of transactions, embodying the core characteristics of DApp.

The operation of a DApp significantly differs from that of traditional internet applications. Traditional internet applications are governed and maintained by a central server, whereas DApps operate in a decentralized network, with each network node potentially serving as a service provider. This attribute

[1][Online]. Avilable: https://figshare.com/s/f44e1399dca60b3672f9

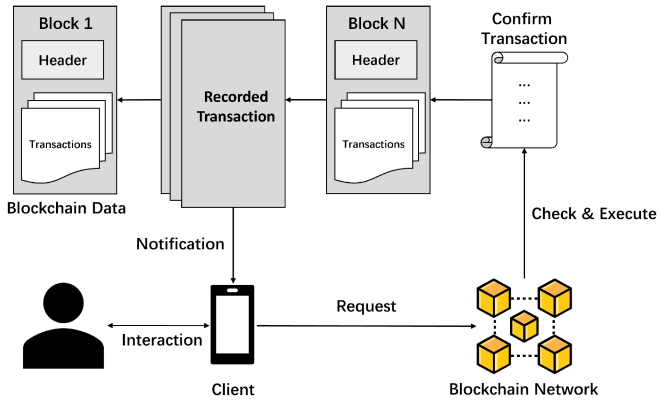

Fig. 2. Simplified process of DApp transaction requests.

renders the entire system more stable and immune to the damage inflicted by any single node. Leveraging blockchain technology, DApps can ensure data integrity and immutability, significantly bolstering user trust. DApp projects exhibit increased complexity, interoperability, and scalability requirements, thus necessitating greater attention during deployment and management. The development of DApps also confronts challenges, including ensuring application performance, managing a large volume of transactions, and preserving user privacy [16]. However, ongoing advancement and refinement in blockchain technology are anticipated to resolve these issues. In summary, DApps constitute a novel application model. Their emergence furnishes us with a more secure, transparent, and efficient service platform, offering boundless possibilities for future application development [15].

III. State Derailment Defects

In this section, we collect relevant data and define derailment defects.

A. Data Collection

We utilize the DAppSCAN [17], an open-source dataset of smart contract defects that contains 25 077 smart contracts from 1139 DApp projects. Considering the financial implications of defects in exchange-type DApps, our study focuses on 46 DApp projects, including 5671 smart contracts.

1) Security Analysis Report: We are thoroughly analyzing 1311 security audit reports from 30 specialized entities. Our analysis identifies a category of defects that we refer to as state derailment defects. These defects occur when essential functions fail to maintain the contract's state. Reports frequently underscore how the outcomes of these functions can affect the contract's state but also reveal cases where they either fail to operate as anticipated or are compromised by other functions. We comprehensively investigate smart contracts with these defects, involving a manual review, rigorous analysis, and experimental evaluation. The audit reports are sourced from entities, such as ChainSecurity [18], runtime verification [19], Quantstamp [20], and Smartdec [21].

2) Data Analysis and Processing: We select DEX projects from the DAppSCAN dataset and exhaustively analyze their corresponding audit reports to mark contracts for defects by cross-referencing them. When dealing with DEX-type smart contracts, we find that these contracts are often complex to compile directly. This is mainly because DEX contracts involve complex transaction logic and dependency on other contracts or libraries. Different solidity versions in one DApp project can cause compilation failures, so it is essential to have strategies for accommodating varying contract versions. In order to compile and process these smart contracts efficiently, we manually process part of the code to ensure that it does not break the original code logic.

Dependency analysis: Before compilation, we analyze the dependencies of the smart contracts in the DEX project. It involves identifying the specific versions of contracts and dependency libraries that the contracts depend on. While compiling, we obtain these dependencies and verify the version compatibility to ensure that the source code can be compiled successfully. Specifically, we would need to supplement the missing dependency libraries manually in some special cases.

Version compatibility: There is a situation where different smart contracts in the same project could have multiple solidity versions. Therefore, it is necessary to match these smart contracts with the compatible compiler version of solidity.

Syntax and functional compatibility: Due to syntactic and functional differences between versions, there will also be discrepant in dependencies, constructors, etc. Therefore, while modifying dependencies, we would make adjustments to the contracts to ensure compatibility. Annotation, during the adjustment process, we do not modify the original semantics and ensure that it is consistent with the defects in the annotation.

B. Defects Definition

In blockchain and smart contract development, safeguarding contract state integrity is critical [22]. State derailment defects constitute a specific category of security defects in smart contracts. Smart contract execution can sometimes result in defects caused by logical inconsistency, design problems, and resource limitations. These defects can cause unauthorized, incorrect, or incomplete updates, changes, or accesses to the system state. Such defects could impact the functionality of a smart contract and may lead to abnormal system operation or exploitation by a malicious user, posing a significant threat to system security [23]. In smart contracts, the state refers to the stored information or variables representing a contract's current state or status at any given moment. Examples of this data may include account balances, ownership details, or any other relevant information the contract may need to execute its functions properly.

Subsequently, we will dissect five defect categories, examining their roots and potential hazards. Each defect signifies a unique smart contract state issue, highlighting the necessity to comprehend and mitigate these for improved contract security.

1) Logical Inconsistencies: Logical inconsistencies in the context of smart contracts refer to a mismatch between the

```
function _withdraw(address _account, uint256
    _withdrawalID) internal override {
        uint256 amount = withdrawLocks.withdraw(
            _account, _withdrawalID);
        livepeer.withdrawStake(_withdrawalID);
        steak.transfer(_account, amount);
        emit Withdraw(_account, amount,
            _withdrawalID);
    }
```

Fig. 3. Example of logical inconsistency defect.

```
function removePool(address pool_address) public
    onlyByOwnerOrGovernance {
        require(frax_pools[pool_address] == true, "
            address doesn't exist already");
        delete frax_pools[pool_address];
        for (uint i = 0; i < frax_pools_array.length
            ; i++){
            if (frax_pools_array[i] == pool_address)
                 {
                frax_pools_array[i] = address(0);
                break;
            }
        }
    }
```

Fig. 4. Example of resource constraint defect.

```
function claimAirdrop(bytes32 calldata proof[]) {
    bool verified = MerkleProof.verifyCalldata(proof
        , merkleRoot, keccak256(abi.encode(msg.
        sender)));
    require(verified, "not verified");
    require(alreadyClaimed[msg.sender], "already
        claimed");
    _transfer(msg.sender, AIRDROP_AMOUNT);
}
```

Fig. 5. Example of access control defect.

```
contract TokenVesting {
    uint256 public initialVestAmount;
    uint256 public vestAmount;
    ...;
    function sendTokens(uint256 _amount) private {
        uint256 vestAmount = _amount;
        if (token.balanceOf(this) < _amount )
        {
            vestAmount = token.balanceOf(this);
        }
        token.transfer(tokenRecepient,vestAmount);
        Vested(vestAmount);
    }
}
```

Fig. 6. Example of type and declaration error defect.

actual logic of the contract code and the designer's expectations or intentions, and this inconsistency usually stems from errors, omissions, or defects in the code implementation [24]. It may manifest itself in incorrect updates to the contract state, feature implementations that do not match business requirements, the creation of security vulnerabilities, or incompatibility with other contracts or blockchain platform standards [25]. This affects the correctness and security of the contract and can lead to loss of assets or failure of contract functionality.

Example: Fig. 3 displays a smart contract withdrawal function. The function neglects to verify the return value of the *transfer* function, which can cause state derailment if the token transfer operation fails but the contract continues executing subsequent operations. This could compromise the system's functionality, leading to user fund losses or the contract's inability to execute the anticipated logic accurately.

2) Resource Constraints: Resource constraints are a series of blockchain resource parameters that constrain the execution of smart contracts, including gas (execution cost), storage space, network bandwidth, and block time [26]. These constraints have a significant impact on the execution of smart contracts. Developers must consider these constraints when writing smart contracts to optimize code, reduce resource consumption, and improve execution efficiency. If a smart contract cannot be completed due to resource constraints, its state may be impaired, affecting its ability to fulfill its obligations.

Example: In Fig. 4, the algorithm aims to remove a specified pool address from an associative array cataloging all pool addresses. The code checks for the pool address's existence, removes it from the mapping and sets its array value to 0×0. Notably, the code risks state derailment due to potential gas overconsumption during array traversal for large-scale arrays.

3) Access Control: Access control is a critical mechanism in smart contracts, which defines which users can access or modify the data in the contract, thus protecting the security of the contract [27]. By accurately setting access rights, access control ensures that only authorized users or participants can operate on contract data, preventing unauthorized access or modification and maintaining the integrity and security of the contract state. However, there is negligence or error in the design or configuration of the access control mechanism. In that case, it may result in unauthorized users or participants being able to modify the contract data, thus triggering problems with the contract state and affecting the normal execution and security of the contract [28].

Example: Fig. 5 exhibits a function handling airdrop claims. It authenticates the claimant's proof and initiates the asset transfer upon successful validation. However, the function neglects to set `alreadyClaimed[msg.sender]` to *true*. This flag prevents users from repeatedly claiming airdrops, but incorrect settings allow for multiple claims, creating security, and abuse issues.

4) Type and Declaration Errors: Type and declaration errors are defects caused by improper variable declarations or incorrect type checking during smart contract development. Such errors are usually detected during the compilation phase, but they can also affect the behavior of the contract at runtime, leading to unexpected state changes. This reduces the security and reliability of smart contracts and exposes them to a high risk of external threats.

Example: Fig. 6 shows a token lock-up contract with a private function called *sendTokens* for sending tokens. Before sending the tokens, the code checks if the balance of tokens in the contract address is sufficient to send the specified amount of tokens.

```
function approve(address _spender, uint256 _value)
    public
    whenNotPaused
    whenUnlocked
    returns (bool)
{
    return super.approve(_spender, _value);
}
```

Fig. 7. Example of exception handling defect.

However, there is a potential defect in this code. The function updates a local variable *vestAmount*, but not the contract's public variable of the same name, which could lead to inconsistencies in the contract's state.

5) Exception Handling: In smart contracts, exception handling is a programming mechanism designed to identify and respond to errors or unintended input data during contract execution. By capturing and handling these exceptions, smart contracts can avoid financial losses, contract malfunctions, state update failures, and even security breaches caused by programming errors or improper external inputs, ensuring the stable operation of the contract and the security of the data.

Example: In the code shown in Fig. 7, if an error occurs while executing `super.approve(_spender, _value)`, it might result in the entire function failing, perhaps leading to the rejection of the entire transaction, which may lead to a denial-of-service attack. Implement robust error handling and fault-tolerance mechanisms to ensure the consistency and integrity of smart contracts.

In essence, state derailment defects exhibit a tendency toward project failures caused by state errors. It underscores vulnerabilities where specific actions lead to the illegal modification of the system's state.

IV. Method

In this section, we introduce the principles of StateGuard, which achieves a contract deconstructor and graph optimizer to detect state derailment defects.

A. Overview

According to Fig. 8, the overall architecture of our approach consists of the following three stages.

1) *Smart contract deconstructor (Section IV-B):* In this stage, we convert the contracts with different versions into AST in JSON format. It adaptively extracts the contract's dependency to reveal the structural and semantic features of contracts.
2) *Graph Optimizer (Section IV-C):* In this stage, we integrate node attributes, dynamically identifies and optimizes, and critical dependency paths. Simultaneously, it decomposes contracts into several subgraphs. Subsequently, it transforms these features from subgraphs into a standardized data format.
3) *Defect Detection (Section IV-D):* In this stage, we feed the standardized data into a GCN for learning potential patterns and features of the graph, and finally identifying the defects.

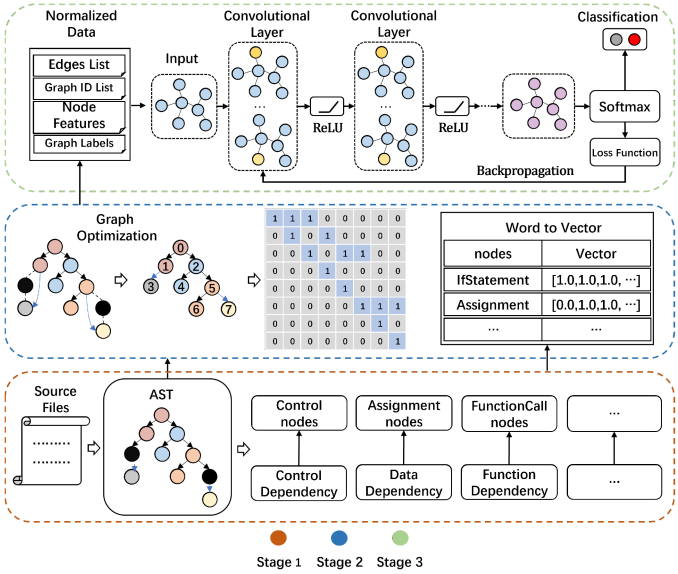

Fig. 8. Data processing workflow of StateGuard.

B. Smart Contract Deconstructor

Smart contract deconstructor is designed to process and analyze smart contract code in DEX. It supports different versions of smart contracts and can handle single-contract and multicontract projects. The core functionality of the smart contract deconstructor is to convert complex smart contract source code into an easy-to-understand and analyze AST representation and then adaptively identify and extract critical features that significantly impact the contract state.

To improve clarity, we will now break down the process into simpler steps, as follows.

Parsing the smart contract: The source code of the smart contract is parsed to generate an AST.

Handling different versions: The generated AST might have different formats depending on the solidity version. The deconstructor adapts to these differences to ensure consistency.

Extracting features: Critical features that impact the contract state are identified and extracted from the AST.

1) Abstract Syntax Tree: AST is a popular program representation paradigm, effectively encapsulating the semantic relationships between program components [29]. It is a hierarchical tree structure that represents the source code's architecture, where each node represents a discrete program segment, including functions, declarations, or expressions. This representation makes it straightforward to analyze and understand the relationships between different parts of the code, such as function calls, variable declarations, and control flow constructs. By clearly visualizing these relationships, we can more easily pinpoint where issues might arise.

ASTs provide multiple abstraction layers—ranging from high-level constructs (e.g., functions, loops) to lower-level details (e.g., expressions, operators). This layered abstraction

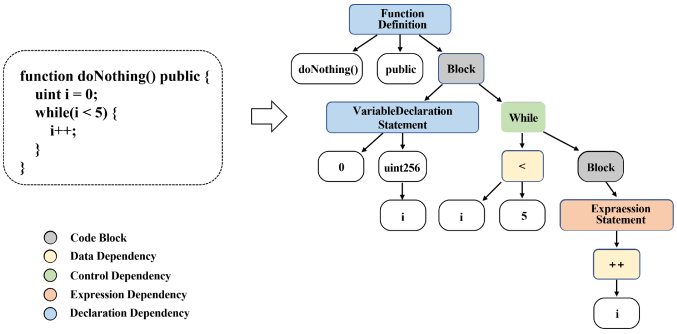

Fig. 9. Example of smart contract source codes' AST.

allows for more granular analysis. For instance, higher level patterns may reveal function-level issues, while lower level patterns can expose intricate bugs in arithmetic operations or conditional statements. Research by Curtis J [30] has proven that converting programs into ASTs that preserve the semantic relationships of program elements allows for better understanding and manipulation of program logic than bytecode. Inspired by this, we convert smart contracts into ASTs and use this structure to perform in-depth analyzes to detect possible security risks and facilitate code optimization.

Fig. 9 shows a source code fragment and the corresponding AST. The tree's root node represents an entire smart contract or function, each internal node represents a statement or expression, and the leaf nodes represent variables, constants, or other basic elements. The AST can represent the types of the various nodes in the tree and their corresponding parts in the source code. Examples include function calls and assignment statements. The tree's depth can indicate the code's complexity, while the number of branches in the tree represents the number of decision points in the code. ASTs allow us to manipulate the source code at a high level. This manipulability is essential for tasks like feature extraction and graph optimization, making the detection process more efficient.

When we use `solc` to compile different versions of solidity smart contracts to generate ASTs, the format of the generated ASTs varies due to the significant differences between solidity versions. For example, in version 0.6 of the AST, the syntax element of a node is identified in the *name* field, while in version 0.8 of the AST, it is identified in the *nodeType* field. In addition, there is a difference in how child nodes are represented. To overcome this challenge, we analyze the AST format. Despite the formatting differences, each node has a unique *id* attribute, and its children are always in an array. Therefore, we look for an array in the current node to determine if a child node exists. Then, we check whether the *id* attribute is present in this array.

When determining the relationship between nodes, we should follow the following guidelines.

1) Nodes within the same hierarchy (i.e., nodes in the same array) are considered siblings.
2) A node in a hierarchy is considered a child of the nearest node in the previous hierarchy (current hierarchy minus one). In other words, if a node is located in a particular level, its parent should be the nearest node in the previous level.

2) Feature Extraction: The syntactic features of defect code can be depicted by suitable data structures, which enable us to fetch source code snippets that match these features [31]. We can identify places with security risks by understanding and analyzing how nodes are connected. Therefore, by traversing the AST, we extract critical features, such as control and data dependencies and assign different roles to different nodes to construct the contract graph. Specifically, we defined five types of critical dependency features based on the syntax elements of solidity.

To clarify, here are the types of critical dependency features.

Declaration dependency: Variable and constant declaration nodes can represent input, output, and state variables in the code. In contrast, function and method declaration nodes can represent the functionality and operations of the smart contract. Discrepancies here can disrupt state updates, leading to potential state derailment.

Expression dependency: Syntax and expression nodes encapsulate the logic and computation. The occurrence of defects could interrupt state updates, resulting in state derailment.

Control dependency: Control dependency nodes define the execution flow of a program, and these operations play an essential role in state management and state changes. Therefore, they are usually highly relevant to defect detection tasks.

Data dependency: Data dependency refers to the dependence of certain program parts on the state or output of other parts. Understanding and tracking data dependencies is critical in identifying and preventing potential security defects.

Function Dependency: Function dependency refers to the possibility that the behavior of a function may depend on other functions. The main focus is on the relationship between functions and how others influence the behavior of one function.

By extracting these critical dependency features from the AST, we provide the necessary structured information for subsequent dynamic path identification and graph representation simplification.

C. Graph Optimizer

Most graph neural networks do not consider the important role that specific nodes play in the network during the information transfer process and instead treat all nodes equally. In addition, the complexity of interactions between DEX contracts leads to the generation of overly large and dense graph representations, which increases the computational complexity and, thus, poses a challenge to the training of graph neural networks. Therefore, to address these issues, we propose a graph optimizer that optimizes the graph representation through, for example, graph attribute integration and dynamic critical path identification to better deal with the performance bottleneck when multiple contracts are encountered.

1) Graph Attribute Integration: An AST is a tree structure representing the source code's abstract syntactic structure. We can construct a directed acyclic graph from the extracted node information by traversing the AST. A graphical structure represents the features extracted from the AST, further enabling complex relationships between features.

Node attribute integration: The importance of node building is that it gives a detailed picture of the network structure, reflecting the relationships between individual nodes. This is important for detecting specific defects because understanding and analyzing how nodes are connected can more accurately identify where security risks may exist. For example, if a node has direct connections to numerous other nodes, it could be a prime target for an attack. Attribute information, such as `id`, `name`, `type`, and `value` of each node is extracted by the smart contract deconstructor. This attribute information is integrated into the node as a node characteristic.

Specifically, we create a set of labels L to store critical dependencies, which can help us focus on and analyze the necessary nodes for executing defect detection tasks. In addition to node extraction, we preserve node attributes, defined as a tuple $(N_{\text{id}}, N_n, N_t, N_v)$, denoted as w, where N_{id} represents the *unique id* of the node in the syntax tree, N_n represents the *name* of the node, N_t represents the *type* of the node, and N_v represents possible *values* that may exist for the node. This attribute provides a straightforward representation of data, facilitating efficient operations and ensuring immutability.

Edge attribute integration: We further construct edges to model the relationships between nodes. Each directed edge represents a possible traversal path between nodes. Specifically, the features of an edge are extracted as a tuple (E_s, E_e, E_t), where E_s and E_e denote its *start* and *end* nodes, respectively, and E_t denotes the *type* of the edge. Such a construction encapsulates the critical information concisely and clearly, which is easy to manipulate and analyze.

2) Graph optimization strategy: After completing the deconstruction of the smart contract, we identify the critical dependency features in the contract and then dynamically analyze the key nodes of the contract and the call relationships between functions. We focus on analyzing those paths with critical dependency features while selectively ignoring those edge paths. By optimizing the dynamically identified critical paths, we simplify the graphical representation by retaining only the critical nodes and connecting edges.

Precisely, our graph optimization process, shown in Fig. 8, consists mainly of removing nodes and edges that do not belong to a predefined list of specific labels, as well as graph simplification by reducing the graph size. We define a specific list of labels L and remove all nodes and edges that do not belong to labels in L. In addition, we further simplify and reduce the size of the graph by traversing the graph and removing specific nodes to improve the processing efficiency. The process of graph optimization can be summarized in the following steps.

1) For each node i in the graph, if its label l_i is not in the list of specific labels L, then it is removed, i.e., the optimized set of nodes $V' = \{i \mid l_i \in L\}$ is obtained.
2) Performs a depth-first traversal of the optimized set of nodes V' to establish parent–child relationships between nodes and to remove the ineligible nodes. Specifically, we define parent_i as the parent of node i and visited as the set of visited nodes.
3) For each node i in the set V', if its label l_i is not in the list of specific labels L and its parent parent_i is not null and has been visited (i.e., it is in visited). If the node i has no child node j, then the node i and its connection edge (parent_i, i) with its parent parent_i are removed. If node i has child j, remove node i, and its connecting edges (parent_i, i) and (i, j) with its parent parent_i and child j, and set the parent of j to parent_i and create a new connecting edge (parent_i, j) with its key children inherit to the parent node.

Algorithm 1: Source Code to Normalized Data.

procedure SOURCETOGRAPH($source\_file$)
  $AST \leftarrow \text{Parse}(source\_file)$
  $word2idx, M \leftarrow \text{Preprocess}(AST)$
  $A, N, V \leftarrow$ ASTtoAdjMatrixAndDict($AST, word2idx$)
  $G \leftarrow \text{OptimizeGraph}(A, N, V)$
  $G' \leftarrow \text{Normalize}(G)$
  return G'
end procedure
procedure ASTTOADJMATRIXANDDICT($AST, word2idx$)
  $A, N, V \leftarrow \text{Initializeemptymatrixanddictionaries}$
  for each $node$ in AST **do**
    $w_i \leftarrow$ node.attributes
    $x_i \leftarrow M[:, word2idx(w_i)]$
    $V[i] \leftarrow x_i$
    for each $neighbor$ in $node.neighbors$ **do**
      $A[i, word2idx(neighbor.label)] \leftarrow 1$
      $N[i] \leftarrow neighbor$
    end for
  end for
  return A, N, V
end procedure
procedure OPTIMIZEGRAPH(A, N, V)
  $V' \leftarrow \text{RemoveNodesAndEdges}(A, N, V)$
  $G \leftarrow \text{DFSAndRemove}(V')$
  return G
end procedure
procedure NORMALIZE(G)
  $G' \leftarrow \text{ApplyTransformations}(G)$
  return G'
end procedure

Through the abovementioned steps, we use the optimized set of nodes and edges as the set of nodes and edges of the final optimized graph. Furthermore, for multicontract projects in DEX, we first identify the nodes that interact between different contracts (i.e., function dependency) and represent these contracts as subgraphs. We then merge these subgraphs into a complete graph. Doing so enables the graph to represent critical dependencies more centrally, thus improving the accuracy (ACC) and efficiency of defect detection.

3) Graph Embedding: Word2Vec [32] is a neural network language model that can transform text data into vector format by learning semantic relationships between words. These vectors can provide valuable inputs for various deep-learning tasks,

including text classification, sentiment analysis, and information retrieval. We convert each graph into adjacency matrices and dictionaries, with node labels as feature vectors. We map node labels into feature vectors by employing the word2idx [33] dictionary and an embedding matrix M. The vector representation of each node i is $v_i = E_{\text{word2idx}}(l_i)$. The topology of a graph $G = (V, E)$ is depicted using an adjacency matrix A. For each node i, an adjacency dictionary N_i is constructed to denote its directly adjacent nodes. In node representation learning, we aim to derive each node's representation vector h_i, mapping each node's feature vector to the representation space via a nonlinear transformation, $h_i = f(X_i)$. This process is crucial to processing graph data, enhancing its applicability and efficiency.

4) Graph Normalization: The trimmed node and edge sets form the refined graph, which is then normalized. Each node attribute w_i is associated with a feature vector $\mathbf{x}_i$ and further processed to normalized vectors $\mathbf{z}_i$. Algorithm 1 outlines the process of source code to normalized data.

The normalized data, including node lists, edge lists, node features, and graph labels, are readied for ensuing deep-learning tasks. Internode relationships are preserved in an edge list $E' = \{(i, j)|A_{ij} = 1, i, j \in V'\}$. Each node's id is stored in a list $G' = \{g_i|i \in V'\}$. Node labels, features, and graph labels are preserved in corresponding lists. The adjacency matrix and dictionary encapsulate the graph's connections. Significantly, we have developed an automated tool to convert source code into normalized data, so the entire process is fully automated.

D. Defect Detection Based on GCN

When dealing with graphical data, the normalized data can act as the input for GCN. GCN [34] is a crucial algorithm for processing graph-structured data, learning, and generating vector representations of nodes by iteratively propagating node features. The key design principles behind our GCN-based approach are as follows:

Graph representation of smart contracts: We represent smart contracts as graphs, where nodes represent syntactic constructs and edges represent syntactic relationships between these constructs. This allows us to leverage GCN's ability to process graph-structured data effectively.

Local connectivity and feature aggregation: GCN excel at capturing local neighborhood information in graphs. By iteratively aggregating information from neighboring nodes, GCN can build rich representations of node contexts, capturing semantics derived from both local and global structures.

Hierarchical information propagation: By stacking multiple convolutional layers, GCN can capture hierarchical information effectively. Each layer captures higher level semantics by combining information from lower level layers, providing a comprehensive understanding of the smart contract code. As shown in Fig. 10, the GCN-based approach effectively captures hierarchical information through multiple convolutional layers. GCN learns node representations by propagating node features in the form of

$$H^{(l+1)} = \sigma(\hat{D}^{-\frac{1}{2}}\hat{A}\hat{D}^{-\frac{1}{2}}H^{(l)}W^{(l)}). \quad (1)$$

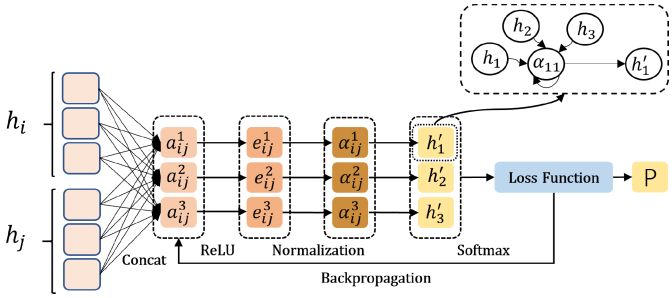

Fig. 10. Graph convolutional neural network.

Each layer utilizes the normalized adjacency matrix with self-loops $\hat{A}$ and the diagonal matrix of degree plus self-loops $\hat{D}$ to propagate node features. Equation (1) updates the node feature matrix $H^{(l+1)}$ by multiplying the weight matrix $W^{(l)}$ and applying the activation function σ. It can be interpreted as weighting the sum of node features and the features of its neighboring nodes and then performing a nonlinear transformation through the activation function.

The output prediction employs the softmax function (2) to make predictions by multiplying the node feature $H^{(L)}$ of the last layer with the weight matrix $W^{(L)}$. We can calculate it as following (2):

$$\hat{y}_i = \text{softmax}\,(H^{(L)}W^{(L)}) \quad (2)$$

where L indicates the last layer and $\hat{y}_i$ denotes the predicted result.

In order to improve the model's performance, we use the back-propagation algorithm [35] along with an optimizer to update the model's parameters. For binary classification problems, the cross-entropy loss function can be used to gauge the difference between the predicted output and the actual label. The model is optimized by minimizing this loss function as following (3):

$$\text{Loss} = -\frac{1}{N}\sum_{i=1}^{N}\left[y_i \log(\hat{y}_i) + (1 - y_i)\log(1 - \hat{y}_i)\right] \quad (3)$$

where y_i represents the actual label of the sample i and $\hat{y}_i$ is the model's prediction for the sample i.

V. Experiment

In this section, we conduct a series of experiments, to evaluate the effectiveness of StateGuard.

A. Experimental Settings

All experiments are executed on a Ubuntu server 22.04 LTS equipped with NVIDIA GeForce GTX 4070Ti GPU, Intel Core i9-13900KF CPU, and 128 G RAM. The software environment includes Python 3.9 and PyTorch 2.0.1.

Regarding the model configuration, we utilize a three-layer GCN, set the adaptive learning rate, and choose the `ReLU` function as the activation function. During model training, we use cross-entropy as the loss function and `Adam` as the optimization algorithm. We use 90% of the dataset for training and

TABLE I
COLLECTED DATASET FOR OUR EVALUATION

Dataset	# Contracts	# Audit reports
DAppSCAN	5,671	1,311
Smartbugs	2,000	0

# Indicates the number of each item.

TABLE II
PERFORMANCE METRICS OF STATEGUARD

Tool	ACC(%)	Recall(%)	Precision(%)	F1(%)	FPR(%)
StateGuard	94.83	94.82	98.28	94.25	0.03

the remaining 10% for validation. In order to comprehensively evaluate the performance of the model on the test set, we select ACC, recall, precision, F1-score, and false positive rate (FPR) as the evaluation metrics.

Specifically, ACC is the ratio of correct predictions to total instances. *Recall* measures how many actual positives we capture. *Precision* reflects how many positives are truly positive. *F1-score* balances Precision and Recall. *FPR* indicates how often negatives are incorrectly identified as positive.

1) Dataset: We use the publicly available DAppSCAN dataset [17] to build a comprehensive dataset critical for identifying and analyzing defects in DApp projects. The dataset includes 703 DApp projects, totalling 23 637 smart contracts, and is continuously updated. We selected 46 DEX projects for analysis, which include a total of 5671 smart contracts. In addition, we collected 1311 security analysis reports from 30 companies or organizations conducting security audits of blockchain technology, smart contracts, and cryptocurrency projects. We thoroughly analyzed the defects in these reports, cross-referencing them with smart contracts for our experimental purposes. We also used another publicly available dataset, Smartbugs [36], a traditional smart contract dataset containing 4285 smart contracts. Table I shows the number of smart contracts we used, containing both vulnerable and benign contracts.

Evaluation metrics: The effectiveness of STATEGUARD is evaluated based on the following research questions (RQs).

- *RQ1:* Is STATEGUARD capable of accurately identifying state derailment defects in the public dataset?
- *RQ2:* Can STATEGUARD find state-related defects undetectable by other tools? How does it compare with existing tools?
- *RQ3:* Can STATEGUARD effectively detect defects in real-world contracts?

B. Answer to RQ1: Defects Detection in a Large-Scale Dataset

To address RQ1, we conduct experiments on 5671 smart contracts from DAppSCAN. We use 90% of these for training and the remaining 10% for testing. The experimental results presented in Table II depict the performance of STATEGUARD, including ACC, recall, precision, F1-score, and FPR. STATEGUARD only identifies whether the contract contains a defect, so we only count it once even if the defect occurs multiple times. As illustrated in Table II, these results substantiate the superior performance of STATEGUARD in detecting state derailment defects.

The experimental outcomes demonstrate the efficiency of STATEGUARD in detecting state derailment defects. The detection ACC of STATEGUARD reaches 94.83%, and the recall rate is 94.82%, indicating that it can accurately identify the most defects. The precision reaches up to 98.28%, showing that most defects identified by STATEGUARD are indeed actual. Moreover, the F1-score is 94.25%, reflecting the comprehensive performance of STATEGUARD. The FPR is only 0.03%, showing that STATEGUARD rarely mislabel standard cases containing defects. It shows that STATEGUARD demonstrates performance in detecting state derailment defects, characterized by high ACC, high recall, and markedly low FPR.

Answer to RQ1. The results indicate that STATEGUARD can identify state derailment defects in the public dataset with considerable accuracy and low false positive rates.

C. Answer to RQ2: Comparison Experiment

In response to RQ2, most detection tools only analyze individual contracts and cannot perform comprehensive detection on the entire project, while multicontracts in DApps are often more complex. This complexity arises from the interaction and dependency between multiple contracts, for which existing tools provide limited support. Furthermore, although some tools can support dependency imports, due to differences between these tools, they require separate adaptation and adjustment for each tool, which is not only time-consuming but also prone to errors. To ensure the validity of the experimental outcomes, we follow the action in [37]. We employ a random sampling strategy to select 2000 smart contracts exhibiting state derailment from the SmartBugs dataset. Concurrently, we gather a series of smart contract defect detection tools from renowned journals and conferences in the fields of software and security (e.g., conference on computer and communications security (CCS) and automated software engineering (ASE)), as well as Mythril [38], which is recommended by the official Ethereum community.

To facilitate comparative analysis, we select eight benchmark smart contract detection tools, i.e., Mythril, Oyente [39], Securify [40], Confuzzius [41], Conkas [42], Manticore [43], Slither [44], and Smartcheck [45]. During the selection process, we consider several factors, as follows:

1) the accessibility of the tool's source code;
2) the tool's capability to detect defects related to the contract state;
3) the tool's support for source code written in solidity;
4) the tool's ability to report the specific locations of potential defect code for manual review.

The experimental results are presented in Table III. As with RQ1, STATEGUARD only identifies whether the contract contains a defect, so we only count it once if the defect occurs multiple

TABLE III
PERFORMANCE COMPARISON OF RELATED TOOLS

Tools	ACC(%)	Recall(%)	Precision(%)	F1(%)	FPR(%)
Mythril	34.89	47.34	50.26	48.75	88.67
Confuzzius	53.43	53.44	66.03	59.07	46.59
Oyente	52.53	50.15	90.67	64.58	32.48
Securify	74.10	56.90	86.74	68.72	8.70
Conkas	74.72	81.10	89.34	85.02	74.13
StateGuard	**91.40**	**90.40**	**92.24**	**91.31**	**7.60**

TABLE IV
DEFECT PROPORTION

Cause of defect	Proportion
Logical inconsistencies	51.15%
Type and declaration errors	12.23%
Resource constraints	20.55%
Exception handling	6.84%
Access control	9.23%

times. Smartcheck indicates that almost all contracts have defects in the dataset. Meanwhile, tools such as Slither and Manticore fail to provide results due to compilation errors, timeouts, or their inability to handle some of the latest versions of smart contracts. In addition, apart from Securify and STATEGUARD, the other tools fail to process all contracts. The analysis results of each tool are filtered, retaining only the valid results to ensure the fairness of the analysis.

In a comparison experiment, STATEGUARD demonstrates performance metrics. Table III shows that it outperforms other detection tools in several essential performance metrics. Notably, STATEGUARD has achieved an ACC rate of 91.40% and a recall rate of 90.40%. At the same time, its Precision is 92.24%, and the F1-score reaches 91.31%, both showing excellent performance. It is particularly noteworthy that the FPR is only 7.60%, significantly reducing the probability of false positives.

The experimental findings indicate that STATEGUARD can discover state-related defects that other tools may miss and exhibit a significant advantage in its overall performance.

Answer to RQ2. The experimental results demonstrate that STATEGUARD can identify unique state-related defects that other tools may overlook, and it can also surpass these tools in several metrics (i.e., accuracy, precision, recall), confirming its efficacy in complex multicontract environments.

D. Answer to RQ3: Real-World Contract Detection

We randomly select 1596 samples of smart contracts from Etherscan [46], which cover smart contracts of different sizes. The sampling methodology we adopt ensures the applicability and validity of our research results.

We run STATEGUARD to detect these real-world smart contracts, and the results show that STATEGUARD successfully identifies smart contracts with state derailment defects. We apply for and obtain CVE certifications for CVE-2023-47033, CVE-2023-47034, and CVE-2023-47035. This means that they are recognized security defects that malicious users could exploit. These defects have been publicized and notified to the vendor. We have also submitted a detailed security audit report to Etherscan that includes the smart contract address with the defect, the exact location of the defect, its property, and the potential impact. It is worth noting that these defects are not successfully detected when using the benchmark tool in RQ2. This indicates that STATEGUARD is more effective in detecting state derailment defects. The advantage of STATEGUARD is its ability to capture the interaction of functions and state variables in smart contracts through graph structures. In addition, GCN can learn the structural features of graphs that are difficult to capture by traditional static analysis or symbolic execution methods.

In summary, STATEGUARD proves its practicality in detecting defects in real-world smart contracts and demonstrates good adaptability to handle smart contracts of various sizes.

Answer to RQ3. STATEGUARD has proven effective in detecting state derailment defects in real-world smart contracts, as evidenced by its successful identification of several novel defects that received CVE IDs, which are undetected by other benchmark tools.

VI. DISCUSSION

In order to understand the state derailment defects in smart contracts, we perform a detailed statistical analysis of the selected DAppSCAN dataset in Table I. As shown in Table IV, we collect and count the percentage of each type of defect in the dataset. As indicated in Table IV, based on the defect ratio analysis, logical inconsistency is the most important source of problems, accounting for 51.15% of the state derailment defects, emphasizing the importance of logical design ACC. Second, type and declaration errors and resource constraints are vital issues affecting contract execution efficiency and security. Exception handling and access control, although accounting for a lower percentage, are equally important in preventing potential security breaches. Therefore, the stability and security of smart contracts are crucial and depend on high-quality code. To ensure this, comprehensive strategies and measures must be implemented.

The practical implications of the discovered state derailment defects in DEXs are profound and multifaceted, impacting both the security and functionality of these platforms. Logical inconsistencies can lead to unintended contract behaviors, potentially resulting in financial losses for users due to erroneous state updates. Resource constraints highlight the necessity for efficient contract design to prevent gas limit overconsumption, which can disrupt contract execution and lead to denial-of-service scenarios. Access control defects expose DEXs to unauthorized state modifications, increasing the risk of fraud and asset theft. Type and declaration errors can cause state inconsistencies, undermining the reliability of the contract's operations and potentially leading to exploitations. Finally, inadequate exception

```
function safeTransferFrom(IERC20Token _token,
    address _from, address _to, uint256 _value)
    public {
   execute(_token, abi.encodeWithSelector(
       TRANSFER_FROM_FUNC_SELECTOR, _from, _to,
       _value));
}
```

Fig. 11. Code snippets of defective contracts.

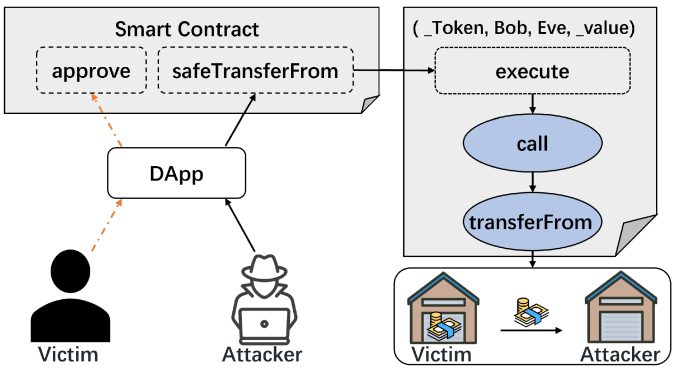

Fig. 12. Illustration of the state derailment case.

handling can result in unhandled errors, causing contract failures and opening avenues for denial-of-service attacks. Collectively, these defects underscore the critical need for rigorous security audits and robust contract design practices to ensure the integrity, reliability, and security of smart contracts within the DEX.

A. Case Analysis

A DEX platform that utilizes smart contracts to enable an autoliquidity mechanism [47] for token transactions. Fig. 11 shows a simplified code snippet of the state derailment defect. We explain how an attacker exploits this defect and highlight the severe consequences that may cause state derailment.

In the ERC20 token standard, the `safeTransferFrom` function is a vital interface to transfer tokens between two addresses. The parameters of this function include the address of the token contract (`_token`), the sender's address (`_from`), the recipient's address (`_to`), and the amount of tokens to be transferred (`_value`). However, this function is defined as public, which means that anyone can call this function, potentially leading to some security issues.

To explain this issue in more detail, we can refer to an example in Fig. 12. The `approve` function is another interface in the ERC20 standard, authorizing other addresses to transfer tokens on behalf of the *victim*. In this case, the *victim* calls the `approve` function and authorizes a certain amount of tokens for the platform's smart contract for trading on their platform. However, the platform's `safeTransferFrom` function does not restrict the identity of callers. Therefore, the *attacker* can exploit this defect by transferring tokens authorized by the *victim* to their address through the platform's smart contract without further consent from the *victim*.

Specifically, the *attacker* effectively operates proxy transfer. Proxy transfer is an operation in the blockchain, especially in the ERC20 token standard, where an authorized third party is allowed us to transfer tokens from one account to another. Since there is no caller authentication within that particular function implementation, this allows successful execution where tokens will be transferred from the *victim* to the *attacker*.

B. Limitations and Future Work

Although we have made progress and optimizations in detecting DEX smart contracts, there remain two limitations.

The time complexity of graphical representation is worth further optimization. DApps are structured with many smart contracts due to their project form. Therefore, the fan-out nature of AST results in a complex graphical representation, even though we have mitigated this problem using graph optimization methods. The time complexity for processing a subgraph is $O(V + E)$, where V is the number of nodes and E is the number of edges. Suppose we must perform complex analysis tasks on the AST, such as data dependency or control dependency analysis. In that case, we may need to traverse the AST multiple times, increasing the time complexity.

Another area for improvement lies in data processing, which involves data extraction and selection. Specific account permissions and call relationships from contracts are critical in data extraction. However, it parses meaningful and highly correlated features from complex structured data. In data selection, noise and outliers widely exist in smart contracts, such as unused variables, dead code, and redundant code, which further increases the complexity of the task. Therefore, meticulous data processing is required to extract and define these features accurately.

Future work could incorporate more domain knowledge into graphical representation optimization and data processing, involving closer collaboration with field experts. Moreover, we plan to explore large language models (LLMs) for smart contract defect detection.

VII. Related Work

A. Defect Detection Tools

Researchers and developers have designed many defect detection tools in response to the potential defects in smart contracts. Solhint [48] is an essential tool for ensuring code quality and consistency in smart contract compliance checking. Static analysis tools [49] (i.e., Mythril, Securify, Slither, and SmartCheck) offer robust defect detection, complemented by dynamic analysis [50] from Manticore Moreover, the comprehensive approach of MythX [51] combines static and dynamic analysis with fuzzing techniques. Deep learning and machine learning have also been utilized, with SaferSc [52] using LSTM networks for defect detection, while Eth2Vec [53] and ContractWard [54] leverage machine learning for defect detection in Ethereum virtual machine bytecode. In addition, VulANalyzeR [55] introduces a novel approach by combining sequential and topological learning through recurrent units and graph convolution, effectively simulating program execution to detect defects.

Meanwhile, TaintGuard [56] stands out as a cross-contract static analysis tool designed to prevent implicit privilege leakage in solidity smart contracts, utilizing taint analysis and instrumentation monitoring to filter call relations for cross-contract calls and detect problematic paths that may lead to privilege leaks.

B. Security Analysis on DEX

A DEX permits users to trade encrypted assets directly via smart contracts, circumventing the need for traditional CEXs. The security of a DEX directly influences the safety of users' assets, thereby necessitating a thorough security analysis. A comprehensive security analysis typically encompasses auditing the DEX's smart contract, assessing the robustness of its design, and simulating attack scenarios [57]. Since the DEX operates on the blockchain, attackers can exploit any security defect, potentially leading to significant financial losses. Therefore, the security of smart contracts in DEX is crucial for the safety of users' assets. For instance, Duan et al. [10] proposed a program analysis technique, VetSC, capable of automatically extracting contract semantics from DApps and performing targeted security checks. VetSC can identify security risks in real-world DApps and ensure the security of DApps. Conversely, Li et al. [11] introduced SolSaviour, a defensive framework designed to repair and recover deployed flawed smart contracts. SolSaviour proposed a novel mechanism, termed the voteDestruct mechanism, which enables contract stakeholders to vote on the destruction of flawed smart contracts. In addition, Xia et al. [3] proposed a method based on the "guilt-by-association" heuristic and machine learning techniques to identify fraudulent tokens on Uniswap from another perspective. This method can accurately label fraudulent behavior on Uniswap. However, from a distinct perspective, Geoffrey et al. [1] introduced SPEEDEX to eliminate the prevalent front-running attacks in CEXs and effectively parallelize transaction processing, achieving high throughput.

VIII. Conclusion

In this article, we present the first systematic study of state derailment defects in DEX smart contracts. We define and classify state derailment defects into five categories and provide examples and detailed analyses for each category. To discover security issues in DEX contracts, we design and develop StateGuard, a deep learning-based framework for detecting state derailment defects in DEX projects. We have evaluated StateGuard on two large datasets, i.e., DAppSCAN and Smartbugs. The results show that StateGuard identifies state derailment defects with 92.24% precision and 90.4% recall, outperforming several existing tools. Furthermore, StateGuard can discover defects in real-world contracts, demonstrating its practicality and effectiveness. As a next step, we plan to explore further leveraging LLM techniques to enhance our defect detection capabilities.

References

[1] G. Ramseyer, A. Goel, and D. Mazières, "SPEEDEX: A scalable, parallelizable, and economically efficient decentralized EXchange," in *Proc. 20th USENIX Symp. Networked Syst. Des. Implementation*, 2023, pp. 849–875.

[2] M. V. Xavier Ferreira and D. C. Parkes, "Credible decentralized exchange design via verifiable sequencing rules," in *Proc. 55th Annu. ACM Symp. Theory Comput.*, 2023, pp. 723–736.

[3] P. Xia et al., "Trade or Trick? Detecting and characterizing scam tokens on uniswap decentralized exchange," *Proc. ACM Meas. Anal. Comput. Syst.*, vol. 5, no. 3, pp. 1–26, 2021.

[4] Medium, "Fixedfloat exploit: Tracing the 26 million lost to the hack," 2024. [Online]. Available: https://medium.com/coinmonks/fixed-float-exploit-tracing-the-26-million-lost-to-the-hack-25fda467b577

[5] T. Chen et al., "Tokenscope: Automatically detecting inconsistent behaviors of cryptocurrency tokens in Ethereum," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2019, pp. 1503–1520.

[6] G. Yu, S. Zhao, C. Zhang, Z. Peng, Y. Ni, and X. Han, "Code is the (f) law: Demystifying and mitigating blockchain inconsistency attacks caused by software bugs," in *Proc. IEEE Conf. Comput. Commun.*, 2021, pp. 1–10.

[7] P. Bose, D. Das, Y. Chen, Y. Feng, C. Kruegel, and G. Vigna, "Sailfish: Vetting smart contract state-inconsistency bugs in seconds," in *Proc. IEEE Symp. Secur. Privacy (SP)*, 2022, pp. 161–178.

[8] M. Ye, Y. Nan, Z. Zheng, D. Wu, and H. Li, "Detecting state inconsistency bugs in Dapps via on-chain transaction replay and fuzzing," in *Proc. 32nd ACM SIGSOFT Int. Symp. Softw. Testing Anal.*, 2023, pp. 298–309.

[9] Y. Liu and Y. Li, "Invcon: A dynamic invariant detector for Ethereum smart contracts," in *Proc. 37th IEEE/ACM Int. Conf. Automated Softw. Eng.*, 2023, pp. 1–4.

[10] Y. Duan et al., "Towards automated safety vetting of smart contracts in decentralized applications," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2022, pp. 921–935.

[11] Z. Li, Y. Zhou, S. Guo, and B. Xiao, "SolSaviour: A defending framework for deployed defective smart contracts," in *Proc. Annu. Comput. Secur. Appl. Conf.*, 2021, pp. 748–760.

[12] Z. Li, W. Guo, Q. Xu, Y. Xu, H. Wang, and M. Xian, "Research on blockchain smart contracts vulnerability and a code audit tool based on matching rules," in *Proc. Int. Conf. Cyberspace Innov. Adv. Technol.*, 2021, pp. 484–489.

[13] G. Wood et al., "Ethereum: A secure decentralised generalised transaction ledger," *Ethereum Project Yellow Paper*, vol. 151, no. 2014, pp. 1–32, 2014.

[14] H. Chen, M. Pendleton, L. Njilla, and S. Xu, "A survey on Ethereum systems security: Vulnerabilities, attacks, and defenses," *ACM Comput. Surv.*, vol. 53, no. 3, pp. 1–43, 2020.

[15] S. Khan, M. Al-Amin, H. Hossain, N. Noor, and M. W. Sadik, "A pragmatical study on blockchain empowered decentralized application development platform," in *Proc. Int. Conf. Comput. Advancements*, 2020, pp. 1–9.

[16] P. Garamvölgyi, Y. Liu, D. Zhou, F. Long, and M. Wu, "Utilizing parallelism in smart contracts on decentralized blockchains by taming application-inherent conflicts," in *Proc. 44th Int. Conf. Softw. Eng.*, 2022, pp. 2315–2326.

[17] Z. Zheng, J. Su, J. Chen, D. Lo, Z. Zhong, and M. Ye, "DAppscan: Building large-scale datasets for smart contract weaknesses in DApp projects," *IEEE Trans. Softw. Eng.*, vol. 50, no. 6, pp. 1360–1373, 2024.

[18] "chainsecurity," 2024. [Online]. Available: https://chainsecurity.com/

[19] "Runtime verification," 2024. [Online]. Available: https://runtimeverification.com/

[20] "Quantstamp," 2024. [Online]. Available: https://quantstamp.com

[21] "Smartdec," 2024. [Online]. Available: https://smartdec.net/

[22] V. Dwivedi, V. Pattanaik, V. Deval, A. Dixit, A. Norta, and D. Draheim, "Legally enforceable smart-contract languages: A systematic literature review," *ACM Comput. Surv.*, vol. 54, no. 5, pp. 1–34, 2021.

[23] T. Sharma, Z. Zhou, A. Miller, and Y. Wang, "A mixed-methods study of security practices of smart contract developers," in *Proc. 32nd USENIX Secur. Symp. (USENIX Secur.)*, 2023, pp. 2545–2562.

[24] F. Tchakounté, K. Amadou Calvin, A. A. A. Ari, and D. J. Fotsa Mbogne, "A smart contract logic to reduce hoax propagation across social media," *J. King Saud Univ.-Comput. Inf. Sci.*, vol. 34, no. 6, pp. 3070–3078, 2022.

[25] M. Barboni, A. Morichetta, and A. Polini, "Smart contract testing: Challenges and opportunities," in *Proc. 5th Int. Workshop Emerg. Trends Softw. Eng. Blockchain*, 2023, pp. 21–24.

[26] N. Ajienka, P. Vangorp, and A. Capiluppi, "An empirical analysis of source code metrics and smart contract resource consumption," *J. Softw.: Evol. Process*, vol. 32, no. 10, pp. 1–22, 2020.

[27] A. Ghaleb, J. Rubin, and K. Pattabiraman, "AChecker: Statically detecting smart contract access control vulnerabilities," in *Proc. 45th IEEE/ACM Int. Conf. Softw. Eng.*, 2023, pp. 945–956.

[28] P. Tolmach, Y. Li, S.-W. Lin, Y. Liu, and Z. Li, "A survey of smart contract formal specification and verification," *ACM Comput. Surv.*, vol. 54, no. 7, pp. 1–38, 2021.
[29] K. Wang, M. Yan, H. Zhang, and H. Hu, "Unified abstract syntax tree representation learning for cross-language program classification," in *Proc. 30th IEEE/ACM Int. Conf. Prog. Comprehension*, 2022, pp. 390–400.
[30] J. Curtis, "On language-agnostic abstract-syntax trees: Student research abstract," in *Proc. 37th ACM/SIGAPP Symp. Appl. Comput.*, 2022, pp. 1619–1625.
[31] F. Al Debeyan, T. Hall, and D. Bowes, "Improving the performance of code vulnerability prediction using abstract syntax tree information," in *Proc. 18th Int. Conf. Predictive Models Data Analytics Softw. Eng.*, 2022, pp. 2–11.
[32] K. W. Church, "Word2Vec," *Natural Lang. Eng.*, vol. 23, no. 1, pp. 155–162, 2017.
[33] D. S. Asudani, N. K. Nagwani, and P. Singh, "Impact of word embedding models on text analytics in deep learning environment: A review," *Artif. Intell. Rev.*, vol. 56, no. 9, pp. 10345–10425, 2023.
[34] D. Yu, Y. Yang, R. Zhang, and Y. Wu, "Knowledge embedding based graph convolutional network," in *Proc. Web Conf.*, 2021, pp. 1619–1628.
[35] Y. Ganin and V. Lempitsky, "Unsupervised domain adaptation by backpropagation," in *Proc. 32nd Int. Conf. Mach. Learn.*, 2015, pp. 1180–1189.
[36] T. Durieux, J. F. Ferreira, R. Abreu, and P. Cruz, "Empirical review of automated analysis tools on 47,587 Ethereum smart contracts," in *Proc. 42nd ACM/IEEE Int. Conf. Softw. Eng.*, 2020, pp. 530–541.
[37] S. Yang, J. Chen, and Z. Zheng, "Definition and detection of defects in NFT smart contracts," in *Proc. 32nd ACM SIGSOFT Int. Symp. Softw. Testing Anal.*, 2023, pp. 373–384.
[38] "Mythril," 2024. [Online]. Available: https://mythril-classic.readthedocs.io/
[39] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making smart contracts smarter," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2016, pp. 254–269.
[40] P. Tsankov, A. Dan, D. Drachsler-Cohen, A. Gervais, F. Buenzli, and M. Vechev, "Securify: Practical security analysis of smart contracts," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2018, pp. 67–82.
[41] C. F. Torres, A. K. Iannillo, A. Gervais, and R. State, "ConFuzzius: A data dependency-aware hybrid Fuzzer for smart contracts," in *Proc. IEEE Eur. Symp. Secur. Privacy*, 2021, pp. 103–119.
[42] N. Veloso, "Conkas: A modular and static analysis tool for Ethereum bytecode," 2023. [Online]. Available: https://github.com/nveloso/conkas/
[43] M. Mossberg et al., "Manticore: A user-friendly symbolic execution framework for binaries and smart contracts," in *Proc. 34th IEEE/ACM Int. Conf. Automated Softw. Eng.*, 2019, pp. 1186–1189.
[44] J. Feist, G. Grieco, and A. Groce, "Slither: A static analysis framework for smart contracts," in *Proc. 2nd IEEE/ACM Int. Workshop Emerg. Trends Softw. Eng. Blockchain*, 2019, pp. 8–15.
[45] S. Tikhomirov, E. Voskresenskaya, I. Ivanitskiy, R. Takhaviev, E. Marchenko, and Y. Alexandrov, "SmartCheck: Static analysis of Ethereum smart contracts," in *Proc. 1st Int. Workshop Emerg. Trends Softw. Eng. Blockchain*, 2018, pp. 9–16.
[46] "Etherscan," 2024. [Online]. Available: https://etherscan.io/
[47] J. Xu, K. Paruch, S. Cousaert, and Y. Feng, "SoK: Decentralized exchanges (DEX) with automated market maker (AMM) protocols," *ACM Comput. Surv.*, vol. 55, no. 11, pp. 1–50, 2023.
[48] "Solhint," 2024. [Online]. Available: https://github.com/protofire/solhint/
[49] T. Yin et al., "An empirical study on implicit constraints in smart contract static analysis," in *Proc. 44th Int. Conf. Softw. Eng.: Softw. Eng. Pract.*, 2022, pp. 31–32.
[50] N. F. Samreen and M. H. Alalfi, "SmartScan: An approach to detect denial of service vulnerability in Ethereum smart contracts," in *Proc. 4th IEEE/ACM Int. Workshop Emerg. Trends Softw. Eng. Blockchain*, 2021, pp. 17–26.
[51] "Mythx," 2024. [Online]. Available: https://mythx.io/
[52] W. J.-W. Tann, X. J. Han, S. S. Gupta, and Y.-S. Ong, "Towards safer smart contracts: A sequence learning approach to detecting security threats," 2018, *arXiv:1811.06632*.
[53] N. Ashizawa, N. Yanai, J. P. Cruz, and S. Okamura, "Eth2vec: Learning contract-wide code representations for vulnerability detection on Ethereum smart contracts," in *Proc. 3rd ACM Int. Symp. Blockchain Secure Crit. Infrastructure*, 2021, pp. 47–59.
[54] W. Wang, J. Song, G. Xu, Y. Li, H. Wang, and C. Su, "Contractward: Automated vulnerability detection models for Ethereum smart contracts," *IEEE Trans. Netw. Sci. Eng.*, vol. 8, no. 2, pp. 1133–1144, Apr.–Jun. 2021.
[55] L. Li et al., "VulANalyzeR: Explainable binary vulnerability detection with multi-task learning and attentional graph convolution," *ACM Trans. Privacy Secur.*, vol. 26, no. 3, pp. 1–25, 2023.
[56] X. Wu, X. Du, Q. Yang, A. Liu, N. Wang, and W. Wang, "TaintGuard: Preventing implicit privilege leakage in smart contract based on taint tracking at abstract syntax tree level," *J. Syst. Archit.*, vol. 141, pp. 102925–102936, 2023.
[57] P. Zheng, Z. Jiang, J. Wu, and Z. Zheng, "Blockchain-based decentralized application: A survey," *IEEE Open J. Comput. Soc.*, vol. 4, pp. 121–133, Mar. 2023.