Penetrating the Hostile: Detecting DeFi Protocol Exploits through Cross-Contract Analysis

Xiaoqi Li, Wenkai Li, Zhiquan Liu, Yuqing Zhang, Yingjie Mao

***Abstract*—Decentralized finance (DeFi) protocols are crypto projects developed on the blockchain to manage digital assets. Attacks on DeFi have been frequent and have resulted in losses exceeding $80 billion. Current tools detect and locate possible vulnerabilities in contracts by analyzing the state changes that may occur during malicious events. However, this victim-only approaches seldom possess the capability to cover the attacker's interaction intention logic. Furthermore, only a minuscule percentage of DeFi protocols experience attacks in real-world scenarios, which poses a significant challenge for these detection tools to demonstrate practical effectiveness. In this paper, we propose DeFiTail, the *first* framework that utilizes deep learning technology for access control and flash loan exploit detection. Through feeding the cross-contract static data flow, DeFiTail automatically learns the attack logic in real-world malicious events that occur on DeFi protocols, capturing the threat patterns between attacker and victim contracts. Since the DeFi protocol events involve interactions with multi-account transactions, the execution path with external and internal transactions requires to be unified. Moreover, to mitigate the impact of mistakes in Control Flow Graph (CFG) connections, DeFiTail validates the data path by employing the symbolic execution stack. Furthermore, we feed the data paths through our model to achieve the inspection of DeFi protocols. Comparative experiment results indicate that DeFiTail achieves the highest accuracy, with 98.39% in access control and 97.43% in flash loan exploits. DeFiTail also demonstrates an enhanced capability to detect malicious contracts, identifying 86.67% accuracy from the CVE dataset. By monitoring existing contracts, we identified five distinct categories of vulnerabilities: repetition abuse, unsafe unintended exploitation, signature violated exploitation, insecure interfaces exploitation, and unrestricted token transfer.**

***Index Terms*—DeFi, Flash loan exploit, Deep learning, Access control**

I. Introduction

Recently, Ethereum-compatible blockchains have witnessed a significant surge in popularity [2]. This increase can be primarily attributed to the growth of ecosystems, such as Decentralized Finance (DeFi) [3], [4] and Non-Fungible Tokens (NFTs) [5], established on these blockchains. These ecosystems, comprising Decentralized Applications (DApps) [6] that execute diverse functions via smart contracts, have garnered a substantial number of active users and market assets. However, security vulnerabilities could exist in any software [7]. As the logical component of DApp, the vulnerability of smart contracts directly affects the security of massive digital assets on users. According to statistics, DeFi-related security incidents have accounted for losses exceeding $80 billion. While access control remains a common vulnerability, the techniques employed to execute such attacks have evolved to become increasingly sophisticated. For example, the attacker found that the verification of the withdraw function within the Orbit Chain contract was inadequate [8]. Through exploiting fake signatures to satisfy the verification threshold, the attacker managed to steal multiple tokens, such as 9,500 ETH and 231 wBTC (with a total value of more than $81.5 million).

Previous research [9], [10], [11] focused on detecting vulnerabilities in smart contracts, utilizing user-defined rules and expert knowledge to standardize detection capabilities. Moreover, the success of deep learning models builds different embeddings, which has been proven in identifying correlation from historical contracts [12], [13], enabling them to discover vulnerable patterns that can capture fragile contracts.

Semantic embedding, which constructs different forms of representation vectors through semantic rules, has been utilized to detect many vulnerabilities. The semantic rules refer to the special rules defined according to the vulnerability or the characteristics of the code, such as the integration of code into control flow based on syntax rules [14], the conversion of code into a tree structure according to call flows [15], and the incorporation of contract transfer into a graph structure corresponding to token flows [16], [17]. These rules construct various vector features by analyzing contract semantics. The approach of semantic embedding has been proven to be helpful for detecting multiple vulnerabilities, such as integer overflow, timestamp dependency, reentrancy, DoS vulnerability, and block state dependency. Initially, embedding processes involved the direct conversion of source code or opcode into distinct representations, followed by using deep learning models to learn sequential textual features [13], [18], [19]. Subsequently, more advanced techniques leverage structured semantic information, such as program paths extracted from Control Flow Graphs (CFG), to enhance detection performance. The graph neural networks [14], [20] or language models [21], [22] are utilized, enabling the classification of code snippets for vulnerability presence. However, it is insufficient for the vulnerability detection of smart contracts. These methods are

Xiaoqi Li, Wenkai Li, Yingjie Mao are with the Hainan University, Haikou, 570228, China (e-mail: csxqli@ieee.org, cswkli@hainanu.edu.cn, yingjiemao@hainanu.edu.cn)

Zhiquan Liu is with the Jinan University, Guangzhou, 510632, China (e-mail: zqliu@jnu.edu.cn)

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

Wenkai Li (cswkli@hainanu.edu.cn) is the corresponding author.

This manuscript is an extended version of our work [1]. It has been extended more than 40% over the WWW conference version, including: (1) Optimization of the background and motivation of the framework (Sec. II) (2) Elaboration on the detailed principles of the framework (Sec. III). (3) Enhancement of the analysis of the experiments (Sec. IV). (4) Addition of discussion with the exploits detected by our framework (Sec. V).

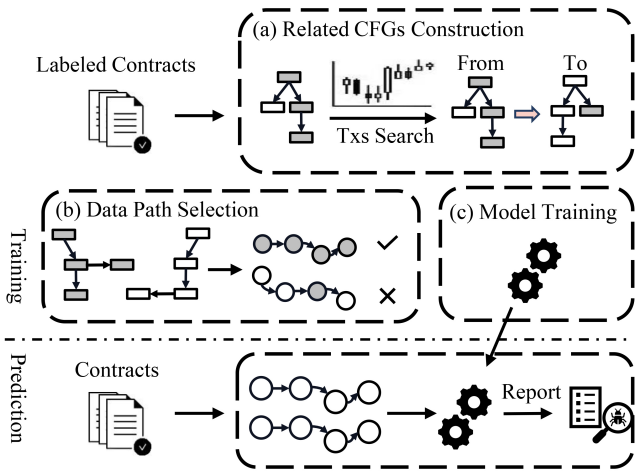

Fig. 1: Overview of DeFiTail. Above the dotted line is the training phase, and below the dotted line is the vulnerability prediction stage.

unaware of capturing accurate attack processes, particularly when multiple contracts are involved, and are incapable of detecting potential vulnerabilities involving multiple contracts. Therefore, there is potential for improvement in the program path representation offered by a single contract when dealing with scenarios that involve multiple contracts.

Deep learning has demonstrated superior performance compared to traditional methods across a wide range of fields. Data-driven could be effectively employed to leverage these advancements, feeding data into a model to learn the implicit features in contracts. However, there are still some challenges in analyzing DeFi projects.

• **Challenge 1 (C1): Invocation Pattern Learning.** Most execution processes in malicious DeFi events involve invocations of multiple contracts [23]. However, the conventional approaches [13], [18], [20] focus on examining isolated data paths or execution paths within individual contracts, which is insufficient for inspecting DeFi protocols that contain lots of invocation patterns in multiple contracts.

• **Challenge 2 (C2): External and Internal Path Unification.** Data paths in DeFi protocols involve transaction-level external paths and code-level internal paths [24], where the external path means the execution flow between contracts, and the internal path means the control flow in a single contract. However, current studies [24], [25], [21], [26] focus on unifying contracts at the transaction level, lacking unifying external paths of DeFi protocols at the code level.

• **Challenge 3 (C3): Data Path Feasibility Validation.** Previous studies [27], [28] have revealed limitations in comprehending the data paths extracted from the CFG. The smart contracts in a DeFi project can generate several CFGs, yielding multiple data paths after the CFGs connection. However, the different choices of entry points can determine different data paths extracted from the connected CFGs, generating different operational sequences. Therefore, it is essential to validate the feasibility of data paths at the operation level.

Our Solution. To address these challenges, we implement DeFiTail, which is a specialized DeFi inspection framework that learns the interaction patterns in data paths with deep learning. As shown in Figure 1, DeFiTail consists of three parts corresponding to the solutions below. Parts (a), (b), and (c) represent the modules S2, S3, and S1 respectively.

• **Solution to C1 (S1)**: We take advantage of sequence and graph learning technology, extracting sequential execution process features and structural heterogeneous graph features. First, we convert the data paths that contain opcodes and operands to a series of sequences according to the execution order in the connected CFGs. Then, a sequence learning model learns the sequence features, and a graph learning model is used to build the heterogeneous graph for extracting the structural features. By combining the sequence and structural features, a more complete feature can be extracted.

• **Solution to C2 (S2)**: We analyze the external transactions in the Ethereum Virtual Machine (EVM), paralleling the internal transaction logic in the smart contract. Through the function segmentation with CFG construction, the 4-byte function signature in the caller function is obtained. Then, we assess whether an invocation exists between functions by examining the presence of the function signatures within another function, unifying external paths and internal paths.

• **Solution to C3 (S3)**: We integrate a symbolic execution stack into DeFiTail, enabling the validation of data path feasibility. The symbolic execution stack uses symbols to record the number of bytes in the stack. Comparing the equal relation between stack height and opcode rules in EVM can determine whether the path is feasible.

The main contributions of this paper are as follows:

- To the best of our knowledge, we propose the first inspection framework DeFiTail, to detect DeFi attacks from the perspective of interaction between attacker and victim. DeFiTail can detect access control and flash loan exploits on various EVM-compatible blockchains (§ III).
- We unify external and internal paths, and connect CFGs between bytecode contracts. Furthermore, the data path validation is formulated into the path reachability problem, identifying feasible data paths (§ III-B).
- We evaluate DeFiTail and it outperforms SOTAs by 16.57% and 11.26% points in detecting access control and flash loan exploits. Moreover, we explore the performance enhancements between CFG connection and data path validation (§ IV).
- We open source DeFiTail at http://doi.org/10.6084/m9.figshare.24117993.

II. Background and Motivation

In this section, we discuss the background of the contract transaction and exploitation definition. We also provide a motivating example.

A. Contract Transaction

Transactions are issued by accounts, Externally Owned Accounts (EOA) and contract accounts, to update the state of the blockchain, e.g., token transfers between two users. Blockchain refers to Ethereum in this paper, one of the most popular blockchain systems. The operation of reversing the

state of the block (e.g., growing balance) is stored in the form of transactions in all blockchain nodes. In EVM, the contract does not directly initiate transactions. EOA account initiates an external transaction, triggering the execution of the logic in the contract. Then the function calls within the contract result in cross-contract interactions (i.e., internal transactions).

An internal transaction records a relationship that occurs within a contract, including calling another contract and creating a new contract. Based on the internal transactions, the EVM converts the logic source contract into deployed instructions, and then performs operations to modify the state of the contract. Through the variability of the state, the correct execution logic in contracts can be guaranteed.

External transaction refers to the action initiated by an EOA. To explore the transaction patterns related to DeFi protocol attacks, we analyze the transactions that interact with the deployed contracts, i.e., the recipient is the contract. Since the contract cannot send transactions, and the contract creation only contains the bytecode of contracts, we focus only on the transactions with contracts.

When a contract interacts with another one by a function call operation, the data flows or call flows are formed. As shown in the Table I, Data flow means that the data execution path follows the sequential execution order. Call flow means the path in a call connection order between two contracts or CFGs. Function calls between two contracts require verification of the function signature through the function selector. Since we focus on EVM-level contracts to ensure adaptability, we analyze the assembly of operation instructions from bytecode. Instructions or opcodes, such as DELEGATECALL and CALLCODE, access the calling contract's storage and check the called contract's signature.

TABLE I: The Terms Definition

Terms	Description
Path	The sequence of opcodes in a specific order.
Data Flow	The sequence of opcodes by the execution order of contracts.
Call Flow	The path in a call connection order between two contracts or CFGs.
Control Flow	The path with the contract execution in a CFG.
DPS Path	The path with the DPS order of the CFG or connected CFG(rCFG).

B. Exploitation Definition

Access Control. Access control refers to programs that fail to effectively manage permission assignments, allowing unauthorized users to access sensitive data or execute critical operations. Attacker A achieves the transaction initiation AT, sending the transactions. Then victim B receives and executes the transaction (i.e., $Tran(B)$). So the Attacker A may exploit vulnerabilities $Defects(\delta[B]_c)$ inherent in the contract code $\delta[B]_c$ of victim B, such as poorly designed role permissions, insufficient checks for functional restrictions, misuse of security libraries, reliance on insecure functions, and inadequate oversight of contract escalation. Then, attacker A can achieve the data access behavior $Access(A, \delta[B]_d)$, and access the sensitive data $\delta[B]_d$ of victim B. The rule of access control exploitation is shown as follows,

$$\frac{AT, Tran(B), Defects(\delta[B]_c), Access(A, \delta[B]_d)}{Access\ Control\ Exploitation}$$

These access control defects can lead to severe consequences, including financial losses, unauthorized disclosure of sensitive information, and unauthorized modifications to contract status.

Flash Loan Exploitation. Flash loan exploitation refers to the use of the flash loan mechanism for malicious purposes. Flash loans are collateral-free loans that enable users to borrow substantial amounts of money within a single transaction, with the requirement to repay the loan before the transaction is finalized. Attacker A can leverage a flash loan to quickly acquire large sums of money, exploiting malicious logic $\delta[A]_c$, such as contract defects and market manipulation, to influence market tokens and price fluctuations, ultimately generating profits after the repayment of the flash loan. The rule of the flash loan attack is shown as follows.

$$\frac{AT, FlashLoan(T), MaliciousLogic(\delta[A]_c), Repayment(T)}{Flash\ Loan\ Exploitation}$$

The key feature of flash loan exploitation is that both borrowing and repayment must occur in a unified transaction.

C. Motivating Example

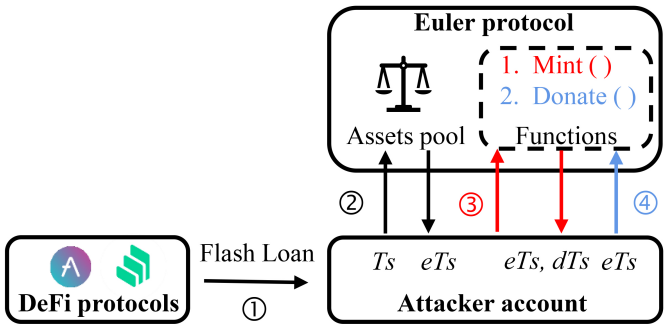

Fig. 2: An Motivating Example of Attacks Process in Euler Protocol. ①-④ steps lead to insolvency of the attacker.

Figure 2 depicts the assault on the Euler protocol, which resulted in a loss of $195 million on March 13, 2023. Accounts include EOAs and contract accounts, and to better describe the overall logic, the attacker contract is represented by the account. Flash loan is a DeFi lending mechanism that permits users with fewer assets and deposits to be collateralized, and the procedure is carried out through smart contracts. In Figure 2, the attacker's contract account borrowed a certain amount of tokens (*Ts*) through a flash loan, and through a series of token conversions with the Euler protocol, it eventually led to bad debt (step ②-④). Among them, *eTs* stands for collateral tokens, and *dTs* stands for debt tokens. In step ②, the attacker deposits *Ts* into the Euler protocol and exchanges a certain amount of *eTs*. Then, by calling the `Mint()` function in step ③, more *eTs* and *dTs* with corresponding proportions are obtained to increase asset liquidity. Finally, the attacker

calls the `Donate()` function in step ④ and pays a certain amount of *eTs* so that the state of *eTs* below *dTs* reaches the liquidation condition. Furthermore, as *eTs* exceed *dTs*, an additional amount remains after repaying the debt. These extra *eTs* seize more collateral than initially intended to repay the flash loan, resulting in a profit for the attacker and a loss in Euler protocol. Therefore, the attack mainly stemmed from failing to check whether the assets were secure in step ③, resulting in the liquidation of the assets after the donation. The rule of the above exploitation is shown as follows,

$$\frac{FlashLoan(T), \exists f \in Func(C), TokenSwap(f), \\ ManipulateState(f), \neg ValidateCollateral(f)}{Flash\ Loan\ Exploitation\ in\ Euler\ Finance}$$

However, the above is only a special attack on the Euler protocol. To correctly learn the specific expression of the threat pattern, and apply the pattern to automatically detect DeFi protocols, our fundamental idea is to leverage the benefits of program analysis and deep learning technologies. We first construct the control flow graph of various contracts, and collect the data flow through program analysis technology. Subsequently, we use deep learning technology to learn threat patterns from the data flows. When analyzing the attack process from the perspective of semantic embedding, it is necessary to focus on the control flows between different contracts, representing the execution logic within the contracts. Due to the fact that the attacks are a dynamic process involving state transitions between multiple contracts. Therefore, we need to deal with the bytecodes in the possible data flow of different contracts through program analysis technology. Furthermore, the patterns of attacks can be learned through deep learning technology, detecting possible attacks on contracts of DeFi protocols. Summarizing these motivations, the program analysis and AI become indispensable in our framework.

III. The Proposed Framework

In this section, we introduce our method, which comprises three key components. First, § III-A transforms the relevant contracts into CFGs and connects them. Next, § III-B extracts the feasible data path by tracking the execution logic of the connected CFG. Finally, § III-C leverages advanced deep learning algorithms to learn and detect the implicit patterns in the data path. Specifically, the CFG is an abstract representation of program execution flow. The data flow is the path through which data flows in a CFG, and we will use the data flow path interchangeably. We divide code into blocks without call relationships, where the block is called a segment and the flow of calls between segments is called call flow. The control flow refers to a path (i.e., the sequence of opcodes during the execution of contracts) in a CFG.

A. Related CFGs Construction

When establishing an internal CFG $G = (N, E)$ within a single contract CA, the initial step involves partitioning basic blocks based on jumping or stopping operations (i.e., `STOP`, `SELFDESTRUCT`, `RETURN`, `REVERT`, `INVALID`, `SUICIDE`, `JUMP`, and `JUMPI`). Note that the basic blocks are represented as the nodes $N \in G$, and the jumping directions mean the edges $E \in G$. Each node $n \in N$ can be represented as a triple-group $n = (id, type, code)$, where the id is the identification, $type$ indicates the type of the node (i.e., starting, ending, and conditional node), $code$ stores the opcodes or operations in the basic block. Subsequently, data flow is orchestrated in alignment with jumping operations (i.e., `JUMP` and `JUMPI`). Specifically, the calldata in the jumping operations contain the function signatures, enabling coherent division by discerning function entries based on varying signatures.

However, constructing a CFG for a single contract faces a challenge. It lacks division of the basic block by calling instructions, disregarding potential interactions involving the invocation of other contracts. The CFG effectively concatenates the sequential execution of all inter-functions. While the external jumping branches exist, the inter-CFG flow can not be completely covered. Specifically, the CFG constructed by single-contract inadequately captures the logic in the multi-contract call scenarios.

Function signatures can be either explicit or implicit. In the case of explicit function signatures, the function signature is stored in the first 4 bytes of the calldata data. This means that we can easily compare whether the hash ID exists in the contract. However, the implicit function signature cannot be ascertained directly, owing to the possibility of the signature changing due to certain operation rules.

Therefore, inspired by the construction of the single-contract CFG [29], [28], we slice the contract snippets with jumping and stopping operations. The calling operations (i.e., `CALL`, `DELEGATECALL`, `STATICCALL`, `CALLCODE`) and the return operations (i.e., `RETURN`) are leveraged as foundational flags to delineate the control flows between contracts. As depicted in Figure 3, we categorize relevant contracts based on their functions and meticulously capture the function signature data for each constituent function throughout the process. When a function involves a calling operation, we divide the function into two segments. The segments are guided by the corresponding instruments (e.g., `CALL`, `DELEGATECALL`, etc.). In the context of the `CALLDATA`, a function signature is located within the call data to which the calling operation is transitioned. In the instance presented in Figure 3, the calling operations are recognized for function α within CFG_t. While the function interface of the function β within CFG_c matches the signature incorporated in the calldata of function α. The execution of steps ① and ② then proceeds to insert a control flow graph node, connecting different CFGs.

To address these challenges, we have implemented several vital discrepancies. As shown in Algorithm 1, we have implemented a function-based method for contract slicing, departing from the traditional basic block segmentation. Moreover, we identify functions culminating with a `RETURN` opcode within the callee contract $\mathbb{B}$, while slicing functions containing the calling operation in the caller function $\mathbb{A}$. Furthermore, the $\perp$ indicates that the function path is feasible, W represents the dictionary-type entry block nodes in CFG, and O means the dictionary that stores the following blocks. Within the instance of $\mathbb{A}$, an inquisition is conducted to establish the presence of the function signatures within $\mathbb{B}$, thereby ensuring the occur-

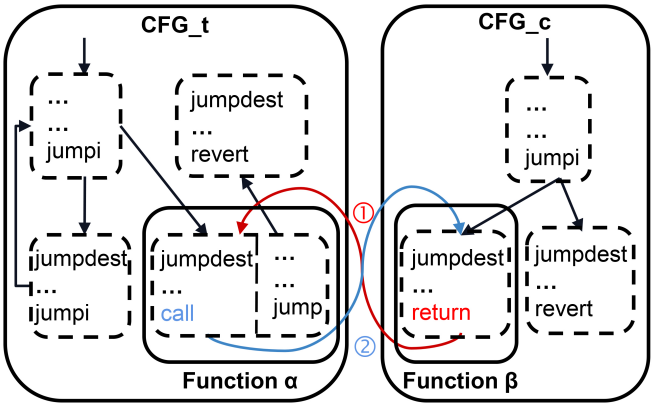

Fig. 3: A Motivating Example of Related CFG Construction. ① and ② represent the sequence of contracts connection.

rence of external calls. This approach offers the advantage of initiating the analysis directly from the function signature of $\mathbb{B}$, thus eliminating exhaustive internal data overhead.

Specifically, we investigate the occurrence of a designated function signature within a specified bytecode contract. Notably, the function signatures are uniquely represented by 4-byte hash IDs. However, a previous study [27] shows that the intricacies inherent exist in the bytecode-to-opcode conversion process (e.g., swap and shift), causing the direct alignment of these function signatures between the bytecode and opcode of the contract is impeded. To address this challenge, we propose a methodology that involves the conversion of bytecodes into opcodes within the context of the Static Single Assignment (SSA) representation. This conversion not only simplifies stack instructions but also elucidates the behavior of the contract. Subsequently, the function signature is extracted from the memory stack through various jump operations. The selection of the function signature directly corresponds to the hash ID present on the stack. During the calldata formation phase, the EVM memory stack stores intermediate values and data from other execution sequences. Distinguishing between static and dynamic callee function parameters may impact the process, leading to variations in EVM generation protocols. The unique eight-character representation of the function signature is readily distinguishable within the EVM stack. Our approach to converting SSA-opcodes is designed to simplify the process of identifying a compatible operand that matches the designated function signature. This innovative process allows us to determine the hash ID within the context denoted as $\mathbb{B}$.

Our investigation focuses on identifying specific contracts involved in transactions that occur at similar timestamps within DeFi protocol events. Within our analysis, in Algorithm 1, we identify the attack contract as the caller in the attack sequence, while the targeted victim contract assumes the callee contracts.

The algorithm makes a prior judgment on lines 5-10, which obtains all function paths that may exit external calls. We cut functions that involve calling operations into distinct code slices., and store them in $\mathbb{A}$ and $\mathbb{B}$, respectively. Then, the function block subgraph containing the precise function signature in the callee is connected to the CFG_t. In line 11, we derive the function signature mapped to the called function, thereby incorporating it as a distinct graph node

Algorithm 1: CFGs Connection For Gathering Data Paths

Input : the caller bytecode contract $contract_t$;
the callee bytecode contract $contract_c$
Output: the connected CFG $rCFG$.

1 $CFG_t \leftarrow$ getCFG($contract_t$);
2 $CFG_c \leftarrow$ getCFG($contract_c$);
3 $F_t \leftarrow$ getFuncPaths(CFG_t); ▷ get the set of function paths;
4 $F_c \leftarrow$ getFuncPaths&Sigs(CFG_c); ▷ get the set of function paths f_c and signatures sig_c;
5 rCFG $\leftarrow CFG_t$
6 **for** $f_t \in F_t$ **do**
7 **if** *isExistCALLs*(f_t) **then**
8 f_p, $f_n \leftarrow$ SplitCALL(f_t); ▷ functions separation;
9 $\mathbb{A}, \mathbb{B} \leftarrow f_p.\perp, f_n.\perp$; ▷ path validation;
10 **end**
11 **end**
12 **for** $f_{op} \in \mathbb{A}$ **do**
13 **if** *isExistSig*(f_{op}, sig_c) **then**
14 **if** *isExistReturn*(f_c) **then**
15 ConnectReachableFunc($\mathbb{A}, \mathbb{B}, f_c$);
16 RemoveUselessConnection($\mathbb{A}, \mathbb{B}$);
17 **end**
18 **end**
19 **end**
20 **return** rCFG ;

within the CFG_t. The final phase of our methodology entails the amalgamation of the derived CFG.

Subsequently, in lines 13-20, we generate distinct actions depending on the presence or absence of the `RETURN`. When `RETURN` exists, we execute the process of connecting external blocks in a manner that parallels the increment of nodes within the graph structure. Precisely, this entails the following steps. Initially, the deletion of the edge linking the function node f_p and its counterpart f_n is performed. Subsequently, a connection is established between the external function and f_n. Conversely, when the `RETURN` is absent, a slightly different procedure ensues. In this scenario, we delete the edge between f_p and f_n, and the edge connecting the external function to f_p. This sequence of actions is concluded by the edges connection between the external function and f_p.

Implementation. The related CFGs construction contains three parts, ① getting CFG from a single contract, ② splitting contracts into basic blocks, and ③ reconnecting the basic blocks. To achieve the first part ①, we construct the CFG from one contract by using the evm-cfg-builder module [30], which has been explored by many works [31], [32], [33], [34]. Then, as for the second part ②, we partition the functions and their corresponding function signatures according to Algorithm 1. As a result, different basic blocks are divided by the CALL operation in the positioning function. Regarding the third part ③, we connect all reachable paths based on the collected function signatures, getting connected CFGs of two contracts.

B. Data Path Selection

Motivation. When analyzing a CFG, the Depth-First Search (DFS) algorithm is utilized to select the data path that embeds as the contract feature. Note that the DFS path (i.e., the path selected with the DFS) could not be exactly equivalent to

the real data path. Since functions can be called at different entries [34], resulting in various data paths that are different from the DFS path. Meanwhile, in § III-A, we employ function calls to establish the CFG connection, which can cause the wrong connection. Therefore, to address the problem, a path validation analysis will be conducted to obtain a more precise data path. After verification, we can identify a precise data path that aligns with the control flow by establishing edges.

As for the symbolic execution technique, we provide a detailed utilization within our framework. Given a data path $S_\pi = \{s_0, s_1, ..., s_n\}$, we define a function P_v to determine the feasibility of the control flow path, which makes a judgment whether it can be executed normally from s_0 to s_n. The primary objective of function P_v is to identify and select all data paths that can be accessed within the graph. Additionally, since we just symbolize the calculation representation to record the height of the symbolic stack, the path explosion issue is also resolved. Thus, there is a scarcity of space-consuming symbols when encountering loops.

A data flow path DP remains feasible, when the transfer condition of control flows on any adjacent blocks is positive. To determine the reachability of a given path, we introduce the $\Gamma(S_\pi)$ in the equation 1, which is a Boolean function to judge the feasibility of the path, the same capability with $\perp$ in the algorithm 1.

$$\Gamma(S_\pi) = \bigwedge_{t=0}^{n-1} \bigvee_{DP \in CF(s_i, s_{i+1})} \bigwedge_{e \in CE(DP)} \Gamma_e(e) \qquad (1)$$

If the $\Gamma(S_\pi)$ returns `true`, the S_π is a true branch, and the ! $\Gamma(S_\pi)$ is the false branch. We define $CF(s_i, s_{i+1})$ to represent the edges on the basic block s_i to s_{i+1} of control flow. Furthermore, $CE(DP)$ encompasses all the control edges within the data path DP, and $\Gamma_e(e)$ signifies the condition at the edge e. The edge is not directed by `JUMPI`, i.e., unconditional jumping, which defaults to true.

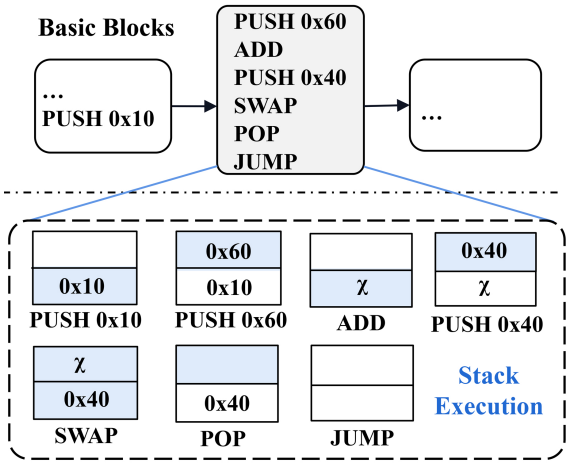

Fig. 4: An Example of Symbolic Stack Execution. χ represents the placeholder of the calculation result, and the blue rectangles mean the operands in or out of the stack for the current step.

Figure 4 depicts a running example of the symbolic stack execution within a function. During this procedure, our primary focus is on the height of the stack, particularly the existence of a target signature during a jumping or calling operation. The evaluation of `ADD` operation results is circumvented by utilizing placeholders to maintain the stack's height. The function signature data includes various operations at the memory stack level, such as `PUSH`, `DUP`, `SWAP`, `AND`, and `POP`. We utilize symbolic execution methodology to manipulate the stack height associated with opcodes, thus allowing us to determine the stack's state during jumping.

In the data paths, we execute each operation in order, and we analyze the symbol stack's height in the last opcode of functions before jumping or calling. In the event that the target opcode lacks sufficient stack height at the moment of its occurrence, it is classified as an infeasible path. In our implementation, each split block is considered a node, and an edge shows the execution direction between blocks. The infeasible path will be removed if the execution of path operations results in an abnormal symbol stack height.

C. Model

Overview. The model contains three parts as depicted in Figure 5. Upon processing the two related contract bytecodes through § III-A and § III-B, the resulting data path is utilized as input for our model. In the first part, we employ the inductive encoder to capture sequence features of the data paths. In the second part, we establish a heterogeneous graph structure for the data paths, treating each data path and opcode as individual nodes, and then learn the graph features. Finally, we merge the features obtained from the previous steps and utilize them for classification in the third part.

The training of the detection model utilized in DeFi protocols necessitates labeled data and the inclusion of all validated paths, encompassing both vulnerable DeFi protocols and secure instances. The model utilizes each validated data path and its corresponding sequence of instructions to learn about various DeFi attack patterns. These paths consist of EVM-compatible contracts in bytecode format and have been verified for feasibility (§ III-B). It first encodes the validated data paths pertaining to each bytecode contract into vectors Vs = $\{\pi_1, \pi_2, ..., \pi_n\}$, and subsequently inputs these vectors into the encoder module to extract hidden features. Finally, a linear and a softmax layer is deployed to aggregate all operations into a fixed-dimensional vector. This vector serves as the foundation for distinguishing whether the protocol is vulnerable or not.

Proposed Architecture Description. Figure 5 illustrates the encoder for processing the data flow paths. The `Pos Emb` means the position embedding, representing the position of tokens in the input sequence. The `Tok Emb` indicates the token embedding, representing the actual tokenized input data. The `Type Emb` represents the different types of tokens in each data path. Specifically, the * in Figure 5 marks the start point and end point of the data path, while the blank signifies the content between these points. Moreover, the `Pos Emb`, `Tok Emb`, and `Type Emb` are the same as BERT [35]. The heterogeneous graph in Figure 5 can be defined as $G = (N, T, E)$, where N denotes different kinds

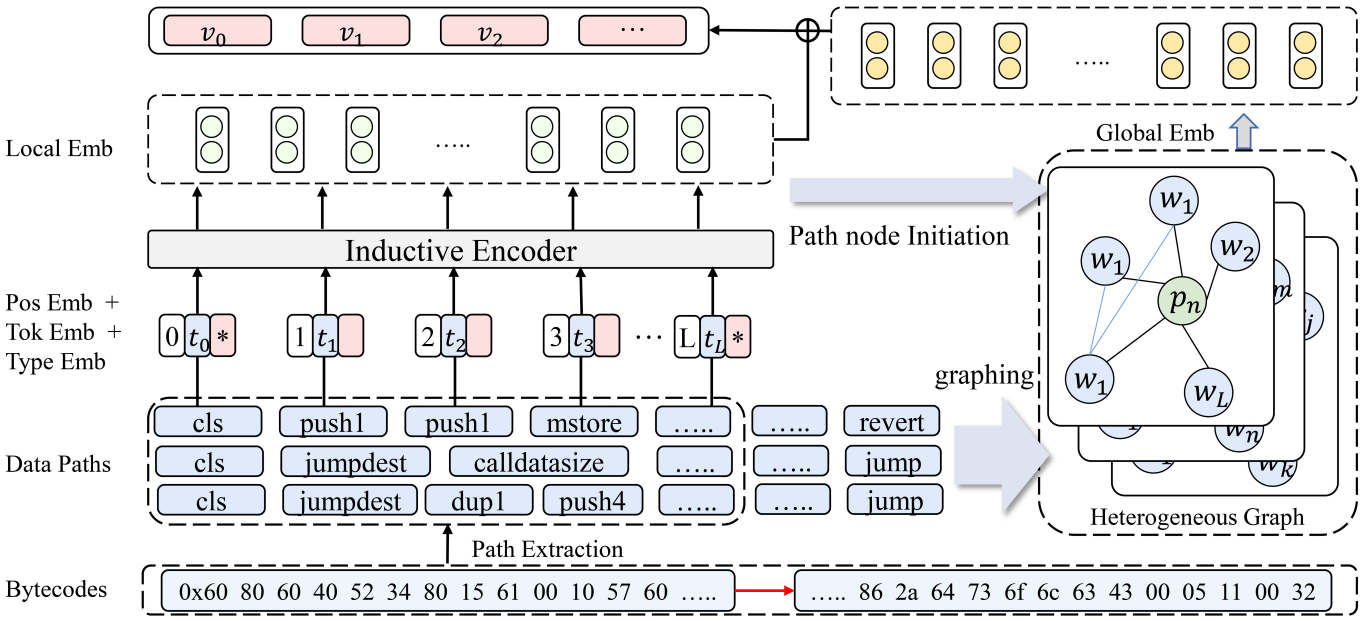

Fig. 5: The Encoder of Data Path Classification Model.

of nodes, and each node $n_i \in N$ has corresponding type $T_i \in T$; T is a type set of nodes, including opcode type w and data path type p; E represents the set of edges connecting nodes. A data path embedding $DP_\pi \in Vs$, which comprises a sequence of n opcode embeddings represented as $\{t_1, t_2, ..., t_n\}$ in Figure 5. Currently, relevant inductive learning methods, such as the transformer, truncate each input to a fixed dimension. Nevertheless, contracts deployed in DeFi entail intricate logic, resulting in generated data paths that are sizeable and use significant computational resources and memory. Consequently, we establish a heterogeneous graph within our corpus to acquire all features ultimately. While the graph structure captures overarching information, it needs to be improved in accessing important details of opcode sequence. Therefore, both inductive (e.g., BERT [35], FastText [36]) and transductive (e.g., GNN [37], GCN [38]) methods are used to proficiently address DeFi contract scenarios, encompassing both position and global graph feature, are utilized.

In the process of constructing a graphical representation from a given data path, we employ the path sequence DP_π to create a node connection matrix. The matrix, as depicted in Figure 5, forms a heterogeneous graph. The black lines connect the path nodes and the corresponding opcode nodes, while the blue lines establish connections among the opcode nodes. We adopt established methodologies to establish these node connections as previously documented in works [38], [39]. Specifically, we utilize the Pointwise Mutual Information (PPMI) and Term Frequency-Inverse Document Frequency (TF-IDF) techniques to determine the weights assigned to edges connecting opcodes and the path-opcode relationships. PPMI, a statistical metric, quantifies the correlation between transactions and is calculated as $PPMI(i,j) = log(p(i,j)/p(i)p(j))$, where $p(i,j)$ represents the probability of co-occurrence of elements i and j. TF-IDF scores clarify the significance of a word within a document, with higher scores signifying a more significant influence.

$$A_{ij} = f(i,j) = \begin{cases} PPMI(i,j) & i,\ j \text{ are opcodes} \\ TF-IDF(i,j) & i \text{ is path},\ j \text{ is opcode} \\ 1 & i = j \\ 0 & \text{otherwise} \end{cases} \tag{2}$$

Initially, we construct a matrix X^a with dimensions of $(n_{path} + n_{opcode}) \times d$, wherein n_{path} refers to the path node, n_{opcode} pertains to the opcode node, and d denotes the dimension of the feature embedding. Based on Equation 2, the weight value is $PPMI(i,j)$ if the two nodes i and j are opcode nodes. In the scenario where i is a path node and j is an opcode node, the edge's weight value is deemed as $TF\text{-}IDF(i,j)$. If i equals to j, then we assign it a weight of 1. Otherwise, it is considered as 0. After completing all traversals, X^a is converted into a triangular matrix to hold the edge weights.

With the weight matrix generation method, we discovered that the section representing the path-path on matrix X^a is 0. It means that the features of the path node itself cannot be learned. To solve it, we utilized the inductive learning method to learn the path feature vector from the opcode sequence and then initialized it into the graph. In this step, we truncate the input path to a fixed length and transform it into a one-hot encoding. Then, we convert it to a vector X^b of dimensions $(n_{path} + n_{opcode}) \times d$, aligning the heterogeneous graph vector. The corresponding dimension of opcode to 0, eliminating the influence of opcode features on graph nodes, i.e., $X^b = \begin{pmatrix} X^a_{path} \\ 0 \end{pmatrix}$. Several studies [22], [15], [40], [41], [42] have demonstrated that BERT models can be fine-tuned to capture specific sequence patterns or behaviors for detecting vulnerabilities in smart contracts. So, we select the BERT structure, which sets 4 layers of transformer encoder, as the

inductive encoder to obtain the feature vector f_{X^b} of each path $DP_\pi = \{t_1, t_2, ..., t_n\}$, which is used to initiate the path nodes in X^a. Simultaneously, we pass the path features through a linear and a softmax layer, demonstrated in Equation 3, to obtain the probability distribution. Note that we omit the bias to simply the explanation.

$$Y_{bert} = softmax(W f_{X^b}) \tag{3}$$

Once the path node initiation is completed on the heterogeneous graph X^g, we utilize a graph convolutional network (GCN) to extract features from all nodes. The GCN generates an output feature X^{gcn} based on Equation 4. We then feed X^{gcn} into a softmax layer to obtain the classification results.

$$X^{gcn} = \sigma(\widetilde{D}^{-\frac{1}{2}}\widetilde{A}\widetilde{D}^{-\frac{1}{2}}X^g W) \tag{4}$$

$$Y_{gcn} = softmax(X^{gcn}) \tag{5}$$

The σ is an activation function, the $\widetilde{D} = \sum_j \widetilde{A}_{ij}$ means the degree of the node i, and the $\widetilde{A}$ is an adjacency matrix that is constructed by the Equation 2. Thus, the $\widetilde{D}^{-\frac{1}{2}}\widetilde{A}\widetilde{D}^{-\frac{1}{2}}$ makes the $\widetilde{A}$ normalization. The $W \in \mathbb{R}^{C\times F}$ indicates a trainable parameters matrix, where C expresses the dimension of the node feature vector, and F represents the output dimension.

$$Y = \lambda Y_{gcn} + (1-\lambda)Y_{bert} \tag{6}$$

Subsequently, a linear interpolation is utilized to merge the inductive and transductive learning results, i.e., the Y_{bert} and Y_{gcn} in Equation 6 are utilized to obtain the final result Y. The $\lambda \in [0, 1]$ controls the balance of two probability distributions.

$$loss(x, i) = W_i(-x_i + log(\sum_j exp(x_j))), \tag{7}$$

where we simply employ the cross entropy loss function $loss(x, i)$ at Equation 7 with weight distributions $W = [W_1, W_2]$ throughout the training, validation, and testing. For the i_{th} category, the W_i is calculated as the total number of categories divided by the number of the i_{th} class, and $x = (x_0, x_1)$ is a non-softmax output. Finally, calculating the weighted sum of the $loss(x, i)$ generated by the two classes, and dividing the sum by the number of classes to obtain the final weighted-cross entropy loss.

IV. Results

In this section, we evaluate the performance of DeFiTail in detecting compromised projects within real-world DeFi protocols at the bytecode level. Specifically, we address the following questions:

- RQ1 **Can DeFiTail outperform other vulnerability detection tools?** We aim to investigate DeFiTail's capability in detecting real DeFi protocols, and how the efficiency compares to state-of-the-art (SOTA) tools.
- RQ2 **How do distinct settings impact the efficiency of DeFiTail?** We aim to examine the individual impact of DeFiTail's each component, and how the effectiveness of related CFG construction and path selection.
- RQ3 **Can our method identify attacks in the real world?** We aim to explore DeFiTail's capability to be applied, or to monitor real-world attacks effectively.
- RQ4 **Can DeFiTail inspect real-world vulnerabilities?** We aim to assess the adaptability of DeFiTail, investigating its capacity to adjust to real-world environments.

A. Experiment Settings

1) Dataset: To evaluate the efficacy of the DeFiTail, we meticulously curated a dataset to facilitate comprehensive comparative evaluations. This dataset comprises a 3,216 substantial collection of 14,301 data paths extracted from 3,216 instances of hacked DeFi events. The data was meticulously gathered from the REKT database [23], one of the largest crypto databases containing DeFi scams, hacks, and exploits. The dataset collection process consists of a total of three filtering criteria. First, we only collect data from blockchains that are compatible with EVM. Second, we only collect events where the loss amount exceeds $10,000. Third, we only collect data with detailed audit reports or attack transaction execution traces, and it must contain the relevant contract address or code. Each incident analysis within our dataset entails an exhaustive examination of malicious accounts and compromised token contracts based on abnormal transactions. Moreover, various attack types are encompassed, including access control, flash loan exploits, honeypots, and rug pull incidents. Significantly, these malicious activities transpired across 25 EVM-compatible blockchains, such as Ethereum, Binance, EOS, Polygon, Arbitrum, Fantom, et al. The dataset is then divided into training and test sets in a 90%: 10% ratio for evaluation. The positive label numbers for access control and flash loan exploitation are 61 and 89. As detailed in Table II, access control and flash loan exploits jointly constitute a mere 4.66% and 5.60% of the entire dataset. Recognizing this imbalance, we implemented an oversampling strategy to rebalance the training set, ensuring adequate exposure to the minority classes.

TABLE II: The Dataset Statistics.

Category	Ratio (%)	Avg Size (byte)
Access Control	4.66	17,349.16
Flash Loan Exploits	5.60	25,641.41

2) Metric and Environment: Since we do not have the ground truth dataset, the presence of a contract solely within a class cannot be definitively determined. Therefore, under our consideration, accuracy is employed as the criterion of evaluation. For instance, consider the scenarios where contract α within a DeFi protocol is revealed to include access control patterns. However, the flash loan exploit that exists within α remains uncertain. Therefore, for a negative sample, we do not know whether access control or flash loan exists on it. Thus, our primary emphasis lies in determining the efficacy of accurately identifying potential threats. All experiments are conducted on a server with an NVIDIA GeForce GTX 4070 Ti GPU and an Intel(R) Core(TM) i7-12700 CPU. The parameter amount of the inductive encoder (i.e., BERT) we utilized is

around 15 million. Regarding graph learning, we use a 2-layer GCN with 200 hidden channels. In addition, we added a dropout of 0.5 to the model to prevent overfitting, which discards 50% of the neurons during the training process.

B. Comparison Analysis (RQ1)

Table III illustrates how DeFiTail can be used to identify and prevent security vulnerabilities in smart contracts, using DeFiTail to detect access control and flash loan exploits. In line with DeFiTail's design objectives, these vulnerabilities arise from the interaction of different contracts. In terms of access control, a contract may be vulnerable to malicious behavior if there has been unintentional manipulation of the contract's permissions. Unauthorized permissions may potentially be granted to other contracts, leading to instigation. Additionally, users have the option to employ the flash loan mechanism to heighten the liquidity of funds, ultimately augmenting conversion between different assets. Given that this process involves many token contracts, Flash Loan incidents may occur when interacting with contracts.

TABLE III: The Comparison Results in Different Tools

Model Name	Access Control	Flash Loan Exploits
Slither[43]	18.67%	–
Maian[44]	45.65%	–
Mythril[45]	64.91%	–
Ethainter[46]	42.86%	–
Achecker[47]	81.82%	–
Midnight[26]	–	86.17%
DeFiTail	98.39%	97.43%

To demonstrate the effectiveness of our framework, we undertake a comparative analysis with SOTAs [45], [46], [47], [26]. As for access control, we conducted a comparison analysis of access control detectors on a single device, including Slither [43], Maian [44], Mythril [45], Ethainter [46], and Achecker [47]. Maian [44] assesses prodigal, suicidal, and greedy contracts, which predominantly concentrates on access control issues arising from token transfers. Slither [43] utilizes a static analysis approach to identify specific operational behaviors, revealing novel behaviors that can bypass permissions. Mythril [45] uses symbolic execution techniques to analyze whether a given input in a program may reach a vulnerable state. Specifically, it uses an SMT (Z3) solver to find out whether a program satisfies the constraints of a vulnerable state. Ethainter [46] uses Datalog to store information flows, and then uses data filtering technology to check modifier-related transactions. Achecker [47] analyzes data dependence through input and state variables, checking conditions of permission control. When using Mythril, specific related vulnerabilities are deemed relevant as access control is not directly detected. In the situations of real DeFi scams, the accuracy in the table will be lower than 64.91%. Since the Ethainter analyzes contracts on three different blockchains, we detected all avaliable contracts and calculated an accuracy of 42.86%. Note that the Ethainter provides a public website[1], the comparison results in Table III are from it. The Achecker, which is a tool that is commonly utilized to identify permission vulnerabilities. After testing, Achecker achieved an accuracy of 81.82%. DeFiTail achieved 98.39% accuracy in the test set, outperforming other SOTA tools in Table III.

As for flash loan exploits, there is limited research on it [48], [49], [50], [26], and the only open-source tool we found is Midnight [26], which is a real-time tool based on event analysis. Midnight associates events with contracts based on real-time transactions, so we obtained the transactions of the test set. Finally, we compare the results of DeFiTail and Midnight in Table III.

Answer to RQ1. DeFiTail exhibits remarkable efficiency in the detection of access control and flash loan exploits, achieving accuracies of 98.39% and 97.43%, respectively.

C. Ablation Analysis (RQ2)

Within DeFiTail, we have evaluated the individual effects of data path selection and CFG connection, and the results are presented in Table IV. Furthermore, the influence of Transformer and heterogeneous graph was investigated.

DeFiTail performs 0.48% and 41.38% for access control better with path selection and related CFGs connection than without them. The detection accuracy of access control attack events is surprisingly improved by path selection. However, the detection performance experiences a significant decline in the absence of a relevant contract to detect the vulnerability jointly. The occurrence arises due to the vulnerability relying heavily on the interaction of different contracts, and it is a critical operation to invoke the malicious contract from other contracts to obtain permissions.

Contrary to access control, models without path related CFG connections perform better than those without. Furthermore, for flash loan events, the results are reversed. The detection of flash loan events is negatively affected by only connecting two related contracts without a path selection process. On the other hand, DeFiTail-np, which only includes the construction of related CFGs, is significantly lower at 12.24% compared to DeFiTail. Due to the minor difference of only 0.21% between DeFiTail and DeFiTail-npc, which lacks both features, it suggests that the connection of two related contracts will enhance the detection results. Therefore, it is evident that the path feasibility verification when connecting related CFGs has a positive impact on the detection results.

TABLE IV: The Ablation Analysis Results. np/npc means the model without path selection/rCFG construction. BERT represents the model with the pre-trained BERT-base.

Model Name	Access Control	Flash Loan Exploits
DeFiTail-np	98.23%	85.19%
DeFiTail-npc	57.33%	97.22%
DeFiTail-LSTM	88.89%	95.00%
DeFiTail-BERT	89.71%	96.30%
DeFiTail	98.39%	97.43%

In our framework, we incorporate path selection, related CFG construction, and contract embedding to extract localized

[1]https://library.dedaub.com/

features during our model design process. Additionally, we construct a heterogeneous graph to capture global features across all contracts. However, we have found that directly utilizing pre-trained models, such as BERT, for classification tasks has been effective. Consequently, we have conducted experiments displayed in Table IV. The LSTM and pre-trained BERT-base models are integrated into our framework, to derive the final classification results in these experiments. Specifically, due to the design of our graph, the two kinds of nodes can be represented. So the graph feature we extracted is the heterogeneous graph feature that captures the distinct features of both opcodes and data paths. Thus, the features acquired by GCN inherently consist of heterogeneous graph features. We then compare these results with the complete DeFiTail approach, which includes one additional global feature beyond DeFiTail-BERT. The results show that access control detection was influenced by different components (LSTM and BERT) and global graph features, which significantly impacted access control detection. However, there is a slight discrepancy in the effectiveness of the three detection techniques in identifying flash loan exploits.

TABLE V: The Results in Signal-Contract Scams Detection.

Model Name	Accuracy(%)
DeFtTail-HoneyPot	60.39
DeFiTail-RugPull	72.51

To further emphasize the limited influence of detecting DeFi events without interaction information, we conducted experiments as outlined in Table V. In the experimental result, we performed CFG construction on a single token contract in honeypot and rug pull scams. Subsequently, we validated the path inaccessibility, followed by model training and experimental testing. The detection accuracy of 60.39% and 72.51% is significantly lower than that of CFG connections, i.e., access control and flash loan exploits.

Answer to RQ2: The presence of both path selection and related CFG connection in access control and flash loan exploits is crucial for the dependencies of DeFiTail.

D. Detection Capability (RQ3)

To verify the effectiveness of DeFiTail in preventing access control scams in the real environment, we evaluate Slither [43], Maian [44], Mythril, SPCon [51], AChecker, and DeFiTail on a CVE dataset listed in Table VI, representing their ability to detect attacks in the real-world environment. Slither [43] tracks variable and state changes in the AST structure to see if inappropriate vulnerability patterns occur. By simulating the execution process of the contract, Maian [44] uses symbolic execution technology to establish and explore the state machine model to check the vulnerability of different states of the contract. SPCon [51] identifies privileged functions that should not have been accessed by mining vulnerabilities in the transaction history.

Table VI indicates that the tag "N/A" denotes inadequate input data necessary for analysis. For example, CVE-2018-19830, CVE-2018-19833, and CVE-2018-19834 are all designated as "N/A" on SPCon because of insufficient transaction data for historical role mining simulations related to access control or permission issues. Additionally, CVE-2018-10705 is marked "N/A" in the DeFiTail score because there are no associated contracts in the malicious events.

TABLE VI: The Evaluation Results on CVE Dataset. N/A means that necessary information is missing.

CVE-ID	Slither	Maian	Mythril	SPCon	AChecker	DeFiTail
CVE-2018-10666	×	×	×	✓	✓	✓
CVE-2018-10705	×	×	×	✓	✓	N/A
CVE-2018-11329	×	×	×	✓	✓	×
CVE-2018-19830	×	×	✓	N/A	✓	✓
CVE-2018-19831	×	×	×	✓	✓	✓
CVE-2018-19832	×	✓	×	✓	✓	✓
CVE-2018-19833	×	×	×	N/A	✓	✓
CVE-2018-19834	×	×	×	N/A	✓	✓
CVE-2019-15078	×	✓	✓	✓	✓	✓
CVE-2019-15079	×	×	×	✓	×	✓
CVE-2019-15080	×	×	×	✓	✓	✓
CVE-2020-17753	✓	×	✓	×	×	✓
CVE-2020-35962	×	×	✓	×	×	✓
CVE-2021-34272	×	×	×	✓	✓	✓
CVE-2021-34273	×	×	×	×	✓	✓

In addition, to ensure maximum detection, we performed vulnerability events using symbolic execution, i.e. Maian and Mythril, for at least 30 minutes on the cases analyzed in Table. VI. Unfortunately, due to a lack of maintenance, we were originally unable to run the SPCon's source code to collect test results. Nevertheless, we have demonstrated the effectiveness of DeFiTail by comparing it to the detection method mentioned in the original paper. This method is considered to be the state-of-the-art tool for detecting real vulnerabilities.

Answer to RQ3: DeFiTail exhibits a remarkable ability to identify security vulnerabilities compared to the state-of-the-art, successfully detecting 86.67% of 15 CVE incidents.

E. Adaptability (RQ4)

To assess the adaptability of DeFiTail in real-world scenarios, we utilized it to monitor the operational contracts of DeFi protocols on the Ethereum blockchain.

However, in the DeFi security environment, false positives can lead to legitimate protocols being incorrectly categorized as malicious activities. For instance, directly profiling by monitoring all accounts that engage with the protocol tends to elevate the false positive rate. Therefore, during the evaluation, we used a strategy aimed at optimizing the false positive rate. Given that transactions on Ethereum follow a power-law distribution—where a small number of accounts account for the majority of transactions—we opted to exclude the top accounts, as identified by Etherscan [2], to mitigate false positives. While this approach may increase the false negative rate, it effectively conserves computing resources.

After a thorough inspection period of 60 hours, we detected several malicious exploits, and the contracts will be recorded

TABLE VII: Descriptions for the 5 Kinds of DeFi Exploits

DeFi exploits	Description
Repetition Abuse	The contract leverages the atomicity of flash loans to allow multiple function calls for a short time, resulting in arbitrage from the liquidity pool.
Unsafe Unintended Exploit	External calls to an unintended address, leading to an unknown account disrupting the normal execution logic and gaining unauthorized access.
Signature Violated Exploit	Message calls invoke functions using a computed 4-byte function signature. When the targeted function does not exist, the failed call leads to unintended behavior.
Insecure interfaces Exploit	When implementing ERC interfaces for DeFi protocols, the block information is utilized in an imprecise manner, resulting in the unauthorized appropriation of token privileges.
Unrestricted Token Transfer	The transfer operation lacks restrictions on the token sender, allowing any individual to withdraw tokens from the contract.

in our online artifact [52]. Since, to the best of our knowledge, none of the previous studies subdivide access control and flash loan exploitation, we refine the found malicious vulnerabilities into 5 categories as shown in Table. VII. Simultaneously, we report the exploits to the CVE repository and the manufacturer.

The initial categories identified in this study consist of two primary dimensions, i.e., access control and flash loan exploits. We conducted a comprehensive examination of the 500 verified contracts over a duration of 60 hours, during which we undertook a detailed manual classification of these contracts into the 5 distinct categories detailed in Table. VII. Specifically, we have reported a number of vulnerabilities, including repetition abuse (CVE-2024-51169), unsafe unintended exploit (CVE-2024-51167, CVE-2024-51172), signature violated exploit(CVE-2024-51170), insecure interfaces exploit (CVE-2024-51171, CVE-2024-51173, and CVE-2024-51174), and unrestricted token transfer (CVE-2024-51168).

Take the repetition abuse in the flash loan exploits as an example, flash loan exploit attacks leverage vulnerabilities in DeFi by borrowing assets from a flash loan agreement. As illustrated by the Oracle contract in Listing 1 and the attacker contract in Listing 2, the exploit extends through a sequence of actions. The attacker initiates the preparation operation by leveraging a flash loan mechanism, as shown in line 9 of Listing 2, to acquire a substantial quantity of stablecoins. Subsequently, during the attacking execution phase in line 14 of Listing 2, the attacker deposits the borrowed assets into Oracle's liquidity pool. Then, the attacker follows an abrupt removal of a significant portion of the liquidity, purposely destabilizing the pool's stability. The resulting imbalance induces volatility in the pool's asset valuation metrics, thereby distorting the price of tokens in the pool. A critical exposure is exploited in line 21 of Listing 2, where the attacker uses the price lookup function `calculateRewards(address user)` in the victim as a view property, which makes the function not need to consume gas. Therefore, the attacker keeps querying the asset prices in the liquidity pool, and when the price in the liquidity pool is inflated and not updated, all the funds in the pool are extracted. Finally, the flash loan repayment is executed.

Distinctions. Previous flash loan attacks, exemplified by the Harvest Finance incident on October 26, 2020 [53], typically involved executing multiple cycles of flash loans to yield several profits through consecutive iterations(i.e., **repeated** cycles of flash loans with **one** profit). However, the situation described in Listing 2 highlights a strategy focused on acquiring small, repeated profits within a singular flash loan transaction(i.e., **one** cycle of a flash loan with **repeated** profits). This strategy seeks to optimize the automated realization of the maximum potential profit from a single flash loan cycle.

```
1 contract Oracle {
2   IPool public pool;
3   mapping(address => uint) public rewards;
4   function calculateRewards(address user) external view
        returns(uint) {
5     uint price = pool.get_virtual_price();
6       return rewards[user] * price;
7     }
8   function claimRewards() external {
9     uint amount = calculateRewards(msg.sender);
10    payable(msg.sender).transfer(amount);
11  }
12 }
```

Listing 1: The Simplified Snippets of an Oracle

```
1 contract FlashLoanAttacker {
2   address pool;
3   address victim;
4   constructor(address _pool, address _victim) {
5     pool = _pool;
6     victim = _victim;
7   }
8   // Initiation
9   function attack(uint256 loanAmount) external {
10    require(msg.sender == owner, "Unauthorized");
11    pool.flashLoan(loanAmount);
12  }
13  // Attacking
14  function executeOperation(uint256 amount) external {
15    // 1. Liquidity Manipulation
16    IERC20(USDC).approve(pool, type(uint).max);
17    IPool(pool).add_liquidity([amount, 0], 0);
18    // 2. Price Fluctuation
19    IPool(pool).remove_liquidity(amount*95/100, [0, 0]);
20    // 3. Repetition Abuse
21    for (uint i = 0; i < 5; ){
22      currentReward = victim.calculateRewards(pool);
23      if (currentReward > amount){ break; }
24      i++;
25    }
26    // 4. Arbitrage
27    victim.claimRewards();
28    // 5. Repayment
29    IERC20(pool).transfer(pool, amount*1001/1000);
30  }
31 }
```

Listing 2: The Simplified Snippets of the Attacker

Answer to RQ4: With DeFiTail, 5 categories of exploits are found, proving its inspection capability in the real world.

V. Discussion

In this section, we will discuss the detected incident, threat to validity, and limitations. For the reason that we detect contracts in bytecodes that are difficult to understand, we demonstrate these in source code, which can improve clarity.

A. Detected Incident

DeFiTail can detect exploitation during the dynamic execution process, even though it uses static techniques. Within the Listing 3, there is an access control issue on line 4. The function `owned()` is declared public, allowing external accounts (i.e., contract accounts and EOAs) to execute it and gain access permissions. It gives the authorization of `owner`, which represents that external accounts can enter contracts or functions by utilizing the `modifier` on line 7, ultimately resulting in a breakdown in access control.

```
1 contract Owned {
2   address public owner;
3   // The key point
4   function owned() public {
5       owner = msg.sender;
6   }
7   modifier onlyOwner {
8       require(msg.sender == owner);
9       _;
10  }
11  function transferOwnership(address newOwner)
        onlyOwner public {
12      owner = newOwner;
13  }
14 }
```

Listing 3: The Simplified Snippets of BTC2X

The attacker creates transactions that call functions that lack access control in order to change the contract creator state. As an example, the attack interaction process with the attacker contract $\delta[A_{att}]$ shown in Listing 4, contract $\delta[A_{own}]$ shown in Listing 3 is deployed at A_{own}, and the attacker's contract $\delta[A_{att}]$ is deployed at A_{att}. At first, the $\delta[A_{att}]$ is passed in the A_{own}. Then, transaction $T = \{A_{att}, A_{own}, owned()\}$ is created, before invoking the $attack()$ function in line 8 of $\delta[A_{att}]$. Then, the A_{own} as the input parameter affects the $msg.sender$ in line 5 of $\delta[A_{own}]$, resulting in the $owner$ changes to attacker A_{att} and controlling the permission. DeFiTail mainly focuses on the transaction process with sensitive data and their interaction styles.

```
1 // Attacker
2 contract Attack {
3   Owned ownedContract;
4   constructor(address _ownedContractAddr) {
5       ownedContract = Owned(_ownedContractAddr);
6   }
7   // attack operation
8   function attack() public {
9       ownedContract.owned();
10  }
11  // transfer the ownership
12  function takeOwnership(address newOwner) public
        {
13      ownedContract.transferOwnership(newOwner);
14  }
15
```

Listing 4: The Attack Snippets to BTC2X

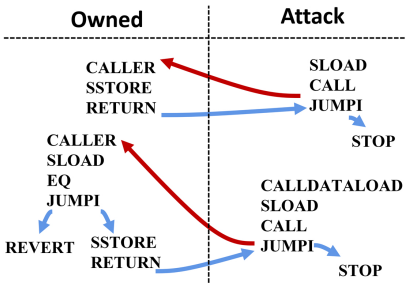

Fig. 6: The Simplified Schematic of the Opcode Flow.

For an attacker who wants to gain access to a victim's contract, it mainly consists of three steps.

In the first step, the attacker executes `CALLDATALOAD` to read the constructor and executes `SSTORE` to store the victim's address into slot (i.e., line 5 of Listing 4).

Figure 6 shows the interaction in the second and third steps. In the second step, the attacker performs `STATICCALL` operation on line 8 of Listing 4 to invoke the `owned()` function of line 4 in the victim contract Listing 3. Then `SLOAD`(to get the slot of the owner variable), `CALLER`(to get the attack contract address), and `SSTORE`(to write the attack contract address to the slot of the owner variable) are executed.

In the third step, the `transferOwnership(address)` function in the victim's contract on line 12 of Listing 4 is called by the attacker, and then the require verification of `JUMPI` jump to line 7 in Listing 3 is performed. This validates `SLOAD`(to read the value of the slot in the owner variable) and `EQ`(to verify that `CALLER` and owner are consistent). Finally, the `SSTORE` instruction (i.e., line 12 in Listing 3) is executed to write the attacker's address to the slot where the owner is located, to obtain all permissions.

B. Runtime Overhead

DeFiTail mainly serves EVM-compliant blockchains, including Ethereum, Binance, EOS, Polygon, Arbitrum, Harmony, Fantom, and et al. While most blockchain platforms claim high transactions per second (TPS), it's crucial to recognize that most of these transactions involve interactions between user wallets rather than with smart contracts.

TABLE VIII: The Comparison of Execution Speed of DeFiTail and Ethereum's TPS

Situation	Tx Amount	Time(s)	TPS
$Ethereum_{all}$	436	30	14.53
$Ethereum_{sc}$	297	30	9.90
DeFiTail	321	19.27	≈16.66

For example, on April 23, 2025, Ethereum shows a reported TPS of 14.7 [2]. As shown in Table VIII, we collect transaction data by randomly selecting a 30-second window, where the transaction amount is 436 and its TPS is 14.53 that are similar to 14.7. During the 30s period, we find that only 297 transactions interacted with smart contracts, with a more accurate TPS of 9.9 for contract interactions.

During the detection process of DeFiTail, we spent an average of 0.06004 seconds per transaction, with a ≈16.66 TPS, which is larger than the TPS in Ethereum.

C. Threat to Validity

Our work focuses on exploring malicious event detection on DeFi protocols.

Internal validity: In the process of gathering the dataset, we meticulously analyzed each security incident to extract the attack and victim contracts, ensuring the dataset's ground truth. Our dataset comprises 1,251 data paths for positive samples, which contains 692 data paths from flash loan attacks and 559 data paths from access control events. In addition, we randomly select 1,336 data paths for negative samples to ensure balance. The dataset is then divided into training and test sets in a 90%: 10% ratio.

External validity: Our dataset encompasses 25 blockchains, making it suitable for various blockchain environments. The contracts for all experiments are gathered from each EVM-compatible blockchain mainnet, and the security events are labeled by the REKT Database, rendering this data highly applicable for evaluation.

D. Limitations

During our evaluation, we lacked an evaluation of the impact and cost of false positives on the DeFiTail framework. Due to the high volume and frequency of transactions on Ethereum, we will involve huge interaction data in the detection process of DeFiTail, which makes it difficult for us to control the false positive rate, which is a disadvantage of DeFiTail. For example, during the 60 hours of monitoring in RQ4, DeFiTail examined about 1.12 million transactions. But to show DeFiTail's performance, we have tested the false negative rate in the testing set. In our results, we achieve a false positive rate of 1.616% for access control and 2.585% for flash loan exploitation.

Moreover, DeFiTail does not show the detailed details of the detected attacks, which makes users need to evaluate the vulnerability logic and its threat model. However, in the detection process, the interaction process involved in each exchange will be detected, which means that the scope of detection is in a transaction, and it is easier to determine whether there is relevant malicious behavior. We will optimize the output report in the future for better understanding.

DeFiTail uses historical data to learn potential exploitation patterns, making it limited to detecting emerging threats that are less frequent. However, through the evaluation of Section IV-E, we find that DeFiTail can also find some new malicious exploitations in the real environment, which belong to access control and flash loan attacks. This illustrates that DeFiTail has the ability to detect new exploitations, which are subcategories of access control and flash loan exploitation.

E. Future Work

Due to design principles and equipment limitations, DeFiTail only focused on the interaction relationship between two contracts on validated transactions. In the future, we will expand DeFiTail from the contracts and transactions aspects. In terms of contracts, we aim to enhance the chain-based interaction analysis capability of DeFiTail across more than three contracts. Regarding transactions, we will deploy full blockchain nodes to shorten the analysis time for pending transaction risks in the memory pool, ensuring that transactions are monitored before being packed into the blocks.

VI. Related Work

In this section, we primarily introduce the relevant work of the paper, which includes the studies on malicious DeFi behavior detection and smart contracts detection and analysis.

A. Malicious DeFi Behavior Detection

The total amount of funds recovered in DeFi has risen to over $77 billion [23], paralleling its rapid growth. During its evolution, DeFi protocols have been incorporated into several well-known blockchains. Smart contracts primarily control the management of users' funds, as these protocols span different blockchains. The interaction between different components creates notable vulnerabilities for potential attacks, which has had a significant impact on DeFi security research. Substantial research efforts [29], [54], [55], [25], [51], [47], [26], [48], [49], [56], [57], [58], [59], [60], [61], [62], [63], [64], [65], [66] have been devoted to attacks on DApps or DeFi projects.

First, there are sorts of studies that focus on the price manipulation or Oracle issues. Q, Kong, et al. [29] propose DeFiTainter, which uses tainted analysis techniques to detect price manipulation vulnerabilities in DeFi protocols. It traces cross-contracts by defining fixed interaction rules from call flow graphs and semantic induction. Z, Zhang, et al. [54] conducted an empirical study examining 516 vulnerabilities in smart contracts between 2021 and 2022. It was found that 34.3% of these vulnerabilities were related to price oracle manipulation. It was also concluded that auditing price oracle vulnerabilities would be more complex, as it would involve other types of real-world vulnerabilities. SecPlF [66] mainly protects Oracle from price manipulation attacks using flash lending, and realizes the protection against arbitrage attacks under the condition of low computational overhead. K, Tjiam, et al. [55] analyze Oracle manipulation events in 2020-2021 using a lifecycle approach and summarize the effectiveness of potential countermeasures. B, Wang [25] proposes Blockeye, a two-step analysis process that combines symbol execution and transaction monitoring to examine Oracle contract state data and detect malicious transactions. J, Xu, et al [64] study the state space model of decentralized exchanges, starting from AMM, analyzing the liquidity problem. J, Xu, et al [65] analyze the algorithmic strategies of DeFi lending markets and liquidity pools, and finds that the volatility of token prices caused by complex contract patterns can cause severe losses.

Y, Liu, et al. [51] proposed SPCon, a solution for checking contracts for access control vulnerabilities. It uses role-mining technology to optimize problem-solving for permission issues. Achecker [47] uses static analysis of data flow and symbols to analyze permission problems from a contract perspective, resulting in improved performance. However, flash credit exploits [26], [48], [49] on many crypto projects, such as Dapp or DeFi projects with complex functionality, have yet to be thoroughly researched. Midnight [26] analyses the actions of events in transactions to detect the presence of flash credit

exploits in specific transactions. D, Wang, et al. [48] and K, Qin, et al. [49] both analyze the real flash loan attack patterns, optimize the efficiency of flash loan expoits, and detect the attack patterns. DeFort [58] framework targets price manipulation attacks in DeFi, which integrates price monitoring and profit calculation mechanisms to achieve attack tracking in different logics. Hyperion [60] utilizes LLM to analyze the DApp front-end pages and identify the inconsistency between them and back-end data through symbolic execution techniques. Both CRPWarner [61] and Defiwarder [63] constructed datalog detection rules for the Rug Pull problem in the DeFi domain, realizing automatic identification of malicious functions. The above methods bring advanced and excellent solutions or unique insights to various security problems in the DeFi field, and they all detect, locate, or defend possible vulnerabilities in the code. Different from them, DeFiTail tries to learn the interaction patterns between attacker and victim contracts in real malicious security events and monitor DeFi protocols.

B. Smart Contract Analysis & Vulnerability Detection

AI has made a significant impact in many industries, including detecting security vulnerabilities in the crypto projects. Various techniques (data process and AI) are being utilized to detect vulnerabilities in smart contracts [10], [20], [14], [18], [22], [67], [21], [11], [68], [69], [70], [71]. Smart contracts can be processed through text sequences, tree structures, or graph structures and then fed into machine learning [10], graph neural networks [20], [14], and inductive neural networks [18], [22], [67] to detect vulnerable smart contracts automatically. W, Chen, et al. [10] analyzed the transaction pattern of Ponzi vulnerabilities between accounts from transactions, then designed features and extracted these features into smart contracts for machine learning. The XGBoost algorithm detects Ponzi vulnerabilities in account and contract features. Z, Liu, et al. [14] embedded the smart contract syntax and integrated expert vulnerability detection capabilities into a graph neural network to detect vulnerabilities. ESCORT [67] first uses a specific encoder to extract the feature information in the contract. It then uses transfer learning with different models to classify the types of vulnerabilities from the features. In addition, various techniques have been combined with AI to detect smart contracts, such as symbolic execution [21] and fuzzing [11]. The ILF [11] exploits the advantages of symbolic execution and fuzzing in deep learning. First, the transaction sequence is obtained through symbolic execution. It is then fed into the GRU model and combined with the fuzz mechanism to achieve improved inference and coverage. SmarTest [21] uses symbolic execution to obtain execution sequences and a well-defined language model to build a corpus that guides another symbolic execution to find vulnerable sequences. With the rise of large language models (LLMs), many studies explore the application of LLMs in the field of smart contract security [68], [69], [70], [71]. The Smartinv [68] is a novel tool that uses multi-modal information to detect vulnerabilities by the Tier of Thought (ToT) prompt strategy. C. Chen, et al. [70] and J. Chen, et al. [71] utilized LLMs to process smart contract vulnerabilities in batches, empirically exploring the role of LLMs in vulnerability detection.

In addition to employing AI methodologies, static and dynamic analysis techniques [72] such as symbolic execution [9], [73], [74], [75], [76], and fuzz testing [77], [78], [79], [80] are widely utilized for the identification of vulnerabilities in smart contracts. These methodologies provide robust frameworks for ensuring the security and integrity of blockchain-based applications. Different from them, DeFiTail uses heterogeneous graphs to capture features in long sequences in a short time, and fuses sequence execution features to obtain optimized contract execution logic features.

VII. Conclusion

The paper proposes DeFiTail, a novel framework for detecting DeFi protocols at the bytecode level. To capture the execution patterns of DeFi protocol attacks, we gather and connect the relevant contracts of the DeFi event attack process with the calling flows. To validate the correctness of CFG, we rely on the symbolic execution stack approach to verify the path feasibility of the connected CFG. To extract global features from extended sequences along the data path of the connected CFG, we construct a graph structure and utilize the BERT model to extract local sequential features. Subsequently, the combined features are used for classification detection. Experimental results reveal that our approach outperforms other SOTA tools in access control and flash loan exploits.

VIII. ACKNOWLEDGMENTS

This work is sponsored by the National Natural Science Foundation of China (No.62362021 and No.62402146), CCF-Tencent Rhino-Bird Open Research Fund (No.RAGR20230115), and Hainan Provincial Department of Education Project (No.HNJG2023-10).

References

[1] W. Li, X. Li, Y. Zhang, and Z. Li, "Defitail: Defi protocol inspection through cross-contract execution analysis," in *Companion Proceedings of the ACM on Web Conference (WWW)*, 2024, p. 786–789.
[2] __, "The ethereum blockchain explorer," 2024, https://etherscan.io/.
[3] S. Werner, D. Perez, L. Gudgeon, A. Klages-Mundt, and et al., "Sok: Decentralized finance (defi)," in *Proceedings of the 4th ACM Conference on Advances in Financial Technologies (AFT)*, 2022, pp. 30–46.
[4] L. Zhou, X. Xiong, J. Ernstberger, S. Chaliasos, Z. Wang, Y. Wang, and et al., "Sok: Decentralized finance (defi) attacks," in *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2023, pp. 2444–2461.
[5] D. Das, P. Bose, N. Ruaro, and et al., "Understanding security issues in the nft ecosystem," in *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2022, pp. 667–681.
[6] H. Singh, "Dapps: Decentralized applications for blockchains," in *Distributed computing to blockchain*, 2023, pp. 87–104.
[7] K. Aggarwal, *Software engineering*. New Age International, 2005.
[8] B. LIU, "$80m lost in first hack of 2024," 2024, https://blockworks.co/news/80-million-lost-orbit-bridge.
[9] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making smart contracts smarter," in *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2016, pp. 254–269.
[10] W. Chen, Z. Zheng, J. Cui, E. Ngai, and et al., "Detecting ponzi schemes on ethereum: Towards healthier blockchain technology," in *Proceedings of World Wide Web Conference (WWW)*, 2018, pp. 1409–1418.
[11] J. He, M. Balunović, N. Ambroladze, P. Tsankov, and M. Vechev, "Learning to fuzz from symbolic execution with application to smart contracts," in *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CSS)*, 2019, pp. 531–548.
[12] J. Chen, "Finding ethereum smart contracts security issues by comparing history versions," in *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2020, pp. 1382–1384.

[13] Z. Gao, V. Jayasundara, L. Jiang, and et al., "Smartembed: A tool for clone and bug detection in smart contracts through structural code embedding," in *Proceedings of the IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2019, pp. 394–397.

[14] Z. Liu, P. Qian, X. Wang, Y. Zhuang, L. Qiu, and X. Wang, "Combining graph neural networks with expert knowledge for smart contract vulnerability detection," *IEEE Transactions on Knowledge and Data Engineering*, vol. 35, no. 2, pp. 1296–1310, 2021.

[15] X. Wang, S. Tian, and W. Cui, "Contractcheck: Checking ethereum smart contracts in fine-grained level," *IEEE Transactions on Software Engineering*, 2024.

[16] B. Yuan, Y. Lu, Y. Fang, Y. Wu, D. Zou, Z. Li, Z. Li, and H. Jin, "Enhancing deep learning-based vulnerability detection by building behavior graph model," in *Proceedings of the IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, 2023, pp. 2262–2274.

[17] S. Wu, Z. Yu, D. Wang, Y. Zhou, L. Wu, H. Wang, and X. Yuan, "Defiranger: Detecting defi price manipulation attacks," *IEEE Transactions on Dependable and Secure Computing*, 2023.

[18] Z. Gao, "When deep learning meets smart contracts," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2020, pp. 1400–1402.

[19] L. T. Li and M. Zhang, "Poster: Eosdfa: Data flow analysis of eosio smart contracts," in *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2022, pp. 3391–3393.

[20] Y. Zhuang, Z. Liu, P. Qian, Q. Liu, X. Wang, and Q. He, "Smart contract vulnerability detection using graph neural networks," in *Proceedings of the 29th International Conference on International Joint Conferences on Artificial Intelligence (IJCAI)*, 2021, pp. 3283–3290.

[21] S. So, S. Hong, and H. Oh, "Smartest: Effectively hunting vulnerable transaction sequences in smart contracts through language model-guided symbolic execution," in *Proceedings of the 30th USENIX Security Symposium (USENIX Security)*, 2021, pp. 17–20.

[22] H. Wu, Z. Zhang, S. Wang, Y. Lei, B. Lin, Y. Qin, and et al., "Peculiar: Smart contract vulnerability detection based on crucial data flow graph and pre-training techniques," in *Proceedings of the International Symposium on Software Reliability Engineering (ISSRE)*, 2021, pp. 378–389.

[23] S. Engineers, "Top crypto hacks," 2023. [Online]. Available: https://de.fi/rekt-database

[24] M. Zhang, X. Zhang, Y. Zhang, and Z. Lin, "Txspector: Uncovering attacks in ethereum from transactions," in *Proceedings of the 29th USENIX Security Symposium (USENIX Security)*, 2020, pp. 2775–2792.

[25] B. Wang, H. Liu, C. Liu, and et al., "Blockeye: Hunting for defi attacks on blockchain," in *Proceedings of the IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, 2021, pp. 17–20.

[26] S. Ramezany and et al., "Midnight: An efficient event-driven evm transaction security monitoring approach for flash loan detection," in *Proceedings of the 20th International Joint Conference on Computer Science and Software Engineering (JCSSE)*, 2023, pp. 237–241.

[27] T. Chen, Z. Li, Y. Zhang, X. Luo, T. Wang, T. Hu, and et al., "A large-scale empirical study on control flow identification of smart contracts," in *Proceedings of the ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 2019, pp. 1–11.

[28] F. Ma, Z. Xu, M. Ren, Z. Yin, Y. Chen, L. Qiao, and et al., "Pluto: Exposing vulnerabilities in inter-contract scenarios," *IEEE Transactions on Software Engineering*, vol. 48, no. 11, pp. 4380–4396, 2021.

[29] Q. Kong, J. Chen, Y. Wang, Z. Jiang, and Z. Zheng, "Defitainter: Detecting price manipulation vulnerabilities in defi protocols," in *Proceedings of the 32st ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, 2023, pp. 531–548.

[30] crytic, "cfg_builder," 2024, https://github.com/crytic/evm_cfg_builder.

[31] Ethersplay, "An open-source protocol for disassembling evm byte code," 2024, https://github.com/crytic/ethersplay.

[32] M. Mossberg, F. Manzano, E. Hennenfent, and et al., "Manticore: A user-friendly symbolic execution framework for binaries and smart contracts," in *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2019, pp. 1186–1189.

[33] D. Guido and et al., "Trail of bits," 2024, https://www.trailofbits.com/.

[34] C. Ferreira Torres, H. Jonker, and R. State, "Elysium: Context-aware bytecode-level patching to automatically heal vulnerable smart contracts," in *Proceedings of the 25th International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 2022, pp. 115–128.

[35] J. D. M.-W. C. Kenton and L. K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," in *Proceedings of the North American Chapter of the Association for Computational Linguistics (NAACL)*, 2019, pp. 1–16.

[36] P. Bojanowski, E. Grave, A. Joulin, and T. Mikolov, "Bag of tricks for efficient text classification," in *Proceedings of the 15th Conference of of the European Chapter the Association for Computational Linguistics (EACL)*, 2017, pp. 427–431.

[37] Z. Wu, S. Pan, F. Chen, G. Long, C. Zhang, and et al., "A comprehensive survey on graph neural networks," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 32, no. 1, pp. 4–24, 2020.

[38] L. Yao, C. Mao, and Y. Luo, "Graph convolutional networks for text classification," in *Proceedings of the AAAI conference on artificial intelligence (AAAI)*, 2019, pp. 7370–7377.

[39] Y. Lin, Y. Meng, X. Sun, Q. Han, K. Kuang, J. Li, and et al., "Bertgcn: Transductive text classification by combining gnn and bert," in *Proceedings of the Findings of the Association for Computational Linguistics (ACL)*, 2021, pp. 1456–1462.

[40] S. Zeng, R. Chen, H. Zhang, and J. Wang, "A high-performance smart contract vulnerability detection scheme based on bert," in *Proceedings of the IEEE 29th International Conference on Parallel and Distributed Systems (ICPADS)*, 2023, pp. 653–658.

[41] S. Hu, Z. Zhang, B. Luo, S. Lu, B. He, and L. Liu, "Bert4eth: A pre-trained transformer for ethereum fraud detection," in *Proceedings of the ACM Web Conference (WWW)*, 2023, pp. 2189–2197.

[42] P. Fang, Z. Zou, X. Xiao, and Z. Liu, "isyn: Semi-automated smart contract synthesis from legal financial agreements," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, 2023, pp. 727–739.

[43] J. Feist, G. Grieco, and A. Groce, "Slither: A static analysis framework for smart contracts," *arXiv preprint arXiv:1908.09878*, 2019.

[44] I. Nikolić, A. Kolluri, I. Sergey, and et al., "Finding the greedy, prodigal, and suicidal contracts at scale," in *Proceedings of the 34th annual computer security applications conference (ACSAC)*, 2018, pp. 653–663.

[45] Mythril, "A security analysis tool for evm bytecode." 2019. [Online]. Available: https://github.com/ConsenSys/mythril

[46] L. Brent, N. Grech, S. Lagouvardos, B. Scholz, and et al., "Ethainter: a smart contract security analyzer for composite vulnerabilities," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2020, pp. 454–469.

[47] A. Ghaleb, J. Rubin, and K. Pattabiraman, "Achecker: Statically detecting smart contract access control vulnerabilities," in *Proceedings of the IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, 2023, pp. 1–12.

[48] D. Wang, S. Wu, Z. Lin, L. Wu, X. Yuan, Y. Zhou, and et al., "Towards a first step to understand flash loan and its applications in defi ecosystem," in *Proceedings of the 9th International Workshop on Security in Blockchain and Cloud Computing (SBC)*, 2021, pp. 23–28.

[49] K. Qin, L. Zhou, B. Livshits, and A. Gervais, "Attacking the defi ecosystem with flash loans for fun and profit," in *Proceedings of the International Conference on Financial Cryptography and Data Security (FC)*, 2021, pp. 3–32.

[50] Quantstamp, "Economic exploit analysis." 2023. [Online]. Available: https://quantstamp.com/economic-exploits

[51] Y. Liu, Y. Li, S.-W. Lin, and C. Artho, "Finding permission bugs in smart contracts with role mining," in *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, 2022, pp. 716–727.

[52] W. Li, "Defitail online artifact," 2024, https://figshare.com/s/3e2eca154a1a66c35225.

[53] Z. Chen, S. M. Beillahi, and F. Long, "Flashsyn: Flash loan attack synthesis via counter example driven approximation," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (ICSE)*, 2024, pp. 1–13.

[54] Z. Zhang, B. Zhang, W. Xu, and Z. Lin, "Demystifying exploitable bugs in smart contracts," in *Proceedings of the IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, 2023, pp. 1–13.

[55] K. Tjiam, R. Wang, H. Chen, and K. Liang, "Your smart contracts are not secure: investigating arbitrageurs and oracle manipulators in ethereum," in *Proceedings of the 3rd Workshop on Cyber-Security Arms Race (CYSARM)*, 2021, pp. 25–35.

[56] Z. Li, X. Peng, Z. He, X. Luo, and T. Chen, "famulet: Finding finalization failure bugs in polygon zkrollup," in *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2024, pp. 971–985.

[57] Z. Li, J. Li, Z. He, X. Luo, T. Wang, X. Ni, W. Yang, X. Chen, and T. Chen, "Demystifying defi mev activities in flashbots bundle," in *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2023, pp. 165–179.

[58] M. Xie, M. Hu, Z. Kong, C. Zhang, Y. Feng, H. Wang, Y. Xue, H. Zhang, Y. Liu, and Y. Liu, "Defort: Automatic detection and analysis of price manipulation attacks in defi applications," in *Proceedings of the*

33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA), 2024, pp. 402–414.
[59] M. Liu, J. H. Huh, H. Han, J. Lee, J. Ahn, F. Li, H. Kim, and T. Kim, "I experienced more than 10 defi scams: On defi users' perception of security breaches and countermeasures," in *Proceedings of the 33rd USENIX Security Symposium (USENIX Security)*, 2024, pp. 6039–6055.
[60] S. Yang, X. Lin, J. Chen, Q. Zhong, L. Xiao, R. Huang, Y. Wang, and Z. Zheng, "Hyperion: Unveiling dapp inconsistencies using llm and dataflow-guided symbolic execution," in *Proceedings of the IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, 2024, pp. 178–190.
[61] Z. Lin, J. Chen, J. Wu, W. Zhang, Y. Wang, and Z. Zheng, "Crpwarner: Warning the risk of contract-related rug pull in defi smart contracts," *IEEE Transactions on Software Engineering*, 2024.
[62] Z. Wu, J. Wu, H. Zhang, Z. Li, J. Chen, Z. Zheng, Q. Xia, G. Fan, and Y. Zhen, "Dappfl: Just-in-time fault localization for decentralized applications in web3," in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, 2024, pp. 137–148.
[63] J. Su, X. Lin, Z. Fang, Z. Zhu, J. Chen, Z. Zheng, W. Lv, and J. Wang, "Defiwarder: Protecting defi apps from token leaking vulnerabilities," in *Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2023, pp. 1664–1675.
[64] J. Xu, K. Paruch, S. Cousaert, and Y. Feng, "Sok: Decentralized exchanges (dex) with automated market maker (amm) protocols," *ACM Computing Surveys*, vol. 55, no. 11, pp. 1–50, 2023.
[65] J. Xu and Y. Feng, "Reap the harvest on blockchain: A survey of yield farming protocols," *IEEE Transactions on Network and Service Management*, vol. 20, no. 1, pp. 858–869, 2022.
[66] S. Arora, Y. Li, Y. Feng, and J. Xu, "Secplf: Secure protocols for loanable funds against oracle manipulation attacks," in *Proceedings of the 19th ACM Asia Conference on Computer and Communications Security (ASIA-CCS)*, 2024, pp. 1394–1405.
[67] C. Sendner, H. Chen, H. Fereidooni, L. Petzi, J. König, J. Stang, and et al., "Smarter contracts: Detecting vulnerabilities in smart contracts with deep transfer learning," in *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2023, pp. 1–18.
[68] S. J. Wang, K. Pei, and J. Yang, "Smartinv: Multimodal learning for smart contract invariant inference," in *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2024, pp. 2217–2235.
[69] Y. Chen, Z. Sun, Z. Gong, and D. Hao, "Improving smart contract security with contrastive learning-based vulnerability detection," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (ICSE)*, 2024, pp. 1–11.
[70] C. Chen, J. Su, J. Chen, Y. Wang, T. Bi, J. Yu, and et al, "When chatgpt meets smart contract vulnerability detection: How far are we?" *ACM Transactions on Software Engineering and Methodology*, vol. 34, no. 4, pp. 1–30, 2025.
[71] J. Chen, Y. Shen, J. Zhang, Z. Li, J. Grundy, Z. Shao, Y. Wang, J. Wang, T. Chen, and Z. Zheng, "Forge: An llm-driven framework for large-scale smart contract vulnerability dataset construction," *Proceedings of the 48th IEEE/ACM International Conference on Software Engineering (ICSE)*, pp. 1–13, 2025.
[72] N. Ivanov, Q. Yan, and A. Kompalli, "Txt: Real-time transaction encapsulation for ethereum smart contracts," *IEEE Transactions on Information Forensics and Security*, vol. 18, pp. 1141–1155, 2023.
[73] P. Bose, D. Das, Y. Chen, Y. Feng, C. Kruegel, and G. Vigna, "Sailfish: Vetting smart contract state-inconsistency bugs in seconds," in *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2022, pp. 161–178.
[74] L. Jin, Y. Cao, Y. Chen, D. Zhang, and S. Campanoni, "Exgen: Cross-platform, automated exploit generation for smart contract vulnerabilities," *IEEE Transactions on Dependable and Secure Computing*, vol. 20, no. 1, pp. 650–664, 2022.
[75] S. Yang, J. Chen, and Z. Zheng, "Definition and detection of defects in nft smart contracts," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, 2023, pp. 373–384.
[76] W. Zhang, Z. Zhang, Q. Shi, L. Liu, L. Wei, Y. Liu, X. Zhang, and S.-C. Cheung, "Nyx: Detecting exploitable front-running vulnerabilities in smart contracts," in *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2024, pp. 2198–2216.
[77] Z. Liu, P. Qian, J. Yang, L. Liu, X. Xu, Q. He, and X. Zhang, "Rethinking smart contract fuzzing: Fuzzing with invocation ordering and important branch revisiting," *IEEE Transactions on Information Forensics and Security*, vol. 18, pp. 1237–1251, 2023.
[78] C. Shou, S. Tan, and K. Sen, "Ityfuzz: Snapshot-based fuzzer for smart contract," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, 2023, pp. 322–333.
[79] W. Chen, X. Luo, H. Cai, and H. Wang, "Towards smart contract fuzzing on gpus," in *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2024, pp. 2255–2272.
[80] R. Liang, J. Chen, C. Wu, K. He, Y. Wu, R. Cao, R. Du, Z. Zhao, and Y. Liu, "Vulseye: Detect smart contract vulnerabilities via stateful directed graybox fuzzing," in *IEEE Transactions on Information Forensics and Security*, vol. 20, 2025, pp. 2157–2170.

Xiaoqi Li is an associate professor at Hainan University. Previously, he was a researcher at the Hong Kong Polytechnic University. He received his Ph.D. in Computer Science from Hong Kong Polytechnic University, MSc in Information Security from the Chinese Academy of Sciences, and BSc in Information Security from Central South University. His current research interests include Blockchain/Mobile/System Security and Privacy, Ethereum/Smart Contract, Software Engineering, and Static/Dynamic Program Analysis. He received best paper awards from INFOCOM'18, ISPEC'17, CCF'18, and an outstanding reviewer award from FGCS'17.

Wenkai Li is currently pursuing a doctor's degree in the School of Cyberspace Security at Hainan University, China. Previously, he received a master's degree in the School of Cyberspace Security at Hainan University. His research lies in smart contract security and malicious behavior analysis, focusing on enhancing blockchain security through software and data analytics. He is also exploring the integration of artificial intelligence, such as graph neural networks and large language models.

Zhiquan Liu received the B.S. degree from the School of Science, Xidian University, Xi'an, China, in 2012, and the Ph.D. degree from the School of Computer Science and Technology, Xidian University, in 2017. He is currently a Full Professor, the Doctoral Supervisor, and the Deputy Dean of the College of Cyber Security, Jinan University, Guangzhou, China. His current research focuses on security, trust, privacy, and intelligence in vehicular networks. He currently serves as the area editor, associate editor, or academic editor of more than 10 SCI-index journals, such as IEEE Transactions on Industrial Informatics, IEEE Internet of Things Journal, Information Fusion, IEEE Network, etc.

Yuqing Zhang is the Director of the Chinese National Computer Network Intrusion Prevention Center, Deputy Director of the Chinese National Engineering Laboratory of Computer Virus Prevention Technology, Vice Dean of the School of Computer and Control Engineering at the Chinese Academy of Sciences, and Professor at Hainan University. He received his Ph.D. from Xi'an University of Electronic Science and Technology. He has presented over 100 papers and 7 national/industry standards. His current research interests include Network Attacks and Prevention, Security Vulnerability Mining and Exploitation, IoT System Security, AI Security, Data Security, and Privacy Protection.

Yingjie Mao is currently pursuing a master's degree in the School of Cyberspace Security at Hainan University, China. Previously, he received a B.E. degree from the Southwest University of Science and Technology. His current research interests include Blockchain Security/Privacy and Large Language Models.