

Peculiar: Smart Contract Vulnerability Detection Based on Crucial Data Flow Graph and Pre-training Techniques

Hongjun Wu[†], Zhuo Zhang^{*}, Shangwen Wang[†], Yan Lei[§], Bo Lin[†], Yihao Qin[†], Haoyu Zhang[‡], Xiaoguang Mao[†]

[†]National University of Defense Technology, Changsha, China

^{*}Guangxi Key Laboratory of Trusted Software, Guilin University of Electronic Technology, Guilin, China

[§]School of Big Data & Software Engineering, Chongqing University, Chongqing, China

[‡]Defense Innovation Institute, Academy of Military Sciences, Beijing, China

{wuhongjun15,wangshangwen13,linbo19,yihaoqin,zhanghaoyu10,xgmao}@nudt.edu.cn,zz8477@126.com,yanlei@cqu.edu.cn

Abstract—Smart contracts with natural economic attributes have been widely and rapidly developed in various fields. However, the bugs and vulnerabilities in smart contracts have brought huge economic losses, which has strengthened people’s attention to the security issues of smart contracts. The immutability of smart contracts makes people more willing to conduct security checks before deploying smart contracts. Nonetheless, existing smart contract vulnerability detection techniques are far away from enough: static analysis approaches rely heavily on manually crafted heuristics which is difficult to reuse across different types of vulnerabilities while deep learning based approaches also have unique limitations. In this study, we propose a novel approach, *Peculiar*, which uses *Pre-training technique* for detection of smart contract vulnerabilities based on *crucial data flow graph*. Compared against the traditional data flow graph which is already utilized in existing approach, crucial data flow graph is less complex and does not bring an unnecessarily deep hierarchy, which makes the model easy to focus on the critical features. Moreover, we also involve pre-training technique in our model due to the dramatic improvements it has achieved on a variety of NLP tasks. Our empirical results show that *Peculiar* can achieve 91.80% precision and 92.40% recall in detecting *reentrancy vulnerability*, one of the most severe and common smart contract vulnerabilities, on 40,932 smart contract files, which is significantly better than the state-of-the-art methods (e.g., Smartcheck achieves 79.37% precision and 70.50% recall). Meanwhile, another experiment shows that *Peculiar* is more discerning to *reentrancy vulnerability* than existing approaches. The ablation experiment reveals that both crucial data flow graph and pre-trained model contribute significantly to the performances of *Peculiar*.

Index Terms—Blockchain, Smart Contract, Vulnerability Detection, Data Flow Graph, Pre-training Techniques

I. INTRODUCTION

Blockchain technology is developing rapidly, having entered a new era dominated by platforms such as Ethereum. The widespread use of smart contract, which is a computerized transaction protocol, is one of the major symbols of blockchain

now. By April 2020, millions of smart contracts have been used in different fields [1], [2], [3], [4]. Since smart contracts in the blockchains always involve cryptocurrencies worthy of millions of USD, bugs within smart contracts often lead to huge amounts of financial losses. For instance, a vulnerability in the Parity Wallet library contract of the standard *multi-sig contract* has been found to freeze hundreds of millions of dollars [5]. Moreover, hackers have great interests in finding vulnerable smart contracts to attack (e.g., 3.6 Million Ether, the Cryptocurrency of Ethereum, was stolen in the DAO incident in 2016 [6]). The immutability of smart contracts makes them extremely difficult to change after they are deployed to the blockchain. As a result, it is critical to detect vulnerabilities among the smart contracts before they are deployed.

So far, developers have developed many effective vulnerability detection tools for smart contracts, such as program analysis based techniques (e.g., SmartCheck [7] and Slither [8]), formal verification based techniques (e.g., ZEUS [9] and Securify [10]), fuzzy testing based techniques (e.g., ContractFuzzer [11]) and symbolic execution based techniques (e.g., Oyente [12], Osiris [13], Mythril [14] and Manticore [15]), etc. Although they are generally effective in detecting smart contract vulnerabilities, they are all based on expert knowledge. That is to say, before applying these approaches, the rules and patterns for detecting the defects must be manually summarized in advance. However, since the number of smart contracts is increasing rapidly, it is impossible to comprehensively enumerate defect patterns existing in all the vulnerable contracts. As a result, the application scenarios of the aforementioned techniques are restricted.

To address this limitation, researchers have proposed various methods to automatically learn the characteristics of smart contract’s defects by utilizing the power of big data. SmartEmbed [16] is such an approach, which uses a deep-learning model to determine whether the detected contract is vulnerable by calculating the similarity with the contracts in a bug database. However, its bug database is in limited scale, with only 52 known buggy smart contracts. Moreover, it performs

^{*} Zhuo Zhang and Shangwen Wang are the corresponding authors. This work is supported by the National Natural Science Foundation of China No.61872445 and No.61672529 as well as the Guangxi Key Laboratory of Trusted Software (Grant/Award Number: kx202008).

bug detection at the statement level of granularity, without taking full advantage of the code structure and semantic information in smart contracts. Zhuang et al. [17] construct a contract graph to represent both syntactic and semantic features of a smart contract function and using graph neural networks (GNNs) for smart contracts. However, we observe that the graph it uses is intricate for representing the program. It not only considers three types of nodes but also involves complex edge information. As we will show in Section II-E, such a complexity makes the model fail to generalize well across different contracts.

In this paper, to address the aforementioned challenges, we propose a novel approach, *Peculiar*, which customizes a Pre-trained model for automatic detection of smart contract vulnerabilities based on crucial data flow graph. In general, *Peculiar* is a learning based technique which does not require manually-defined templates. It, however, mainly embodies two key novelties. First, unlike Zhuang et al. [17] representing program by explicitly detailed graph information, *Peculiar* only captures crucial data flow information which preserves enough features of a program and at the same time enables the model to generalize well across different contracts. Second, to better represent our extracted graph, we embed a pre-trained model in this approach. The intuition is that pre-trained models in natural language processing (NLP) have promoted the development of diverse software engineering tasks [18], [19], [20]. We are thus motivated to utilize potential benefits from pre-trained graph model for smart contract vulnerability detection.

We have conducted extensive experiments on over 200k real-world smart contracts in the **SmartBugs Wild Dataset** [21], which is a recently-released and large-scale dataset for smart contract vulnerabilities. The results show that our approach significantly and consistently outperforms state-of-the-art methods in detecting *reentrancy vulnerability* which is one of the most severe and common smart contract vulnerabilities [22], [23], where the precision and recall reached 91.80% and 92.40% respectively, which are 5.26% and 11.82% higher than the highest values among the existing methods, leading to an overall F1 value of 92.10%.

In summary, the main contributions of this paper are:

- We propose a new approach for smart contract representation which is based on crucial data flow information. Such an approach can capture the key features of a contract while avoid being overfitting to detailed information.
- We implement our approach as *Peculiar* and apply it on the smart contract vulnerability detection task. To our best knowledge, *Peculiar* is the first to leverage the pre-training technique on this task.
- We perform extensive experiments and demonstrate that *Peculiar* can achieve significantly better performance than the state-of-the-art approaches on detecting *reentrancy vulnerability*.

II. BACKGROUND AND MOTIVATION

A. Smart Contract Vulnerability Classification

Attention is increasingly being focused on the security of smart contracts. The existing researches, however, have not proposed a unified classification for smart contract vulnerabilities. The detection tools have different classifications of vulnerabilities depending on the understandings of their authors. For example, ContractFuzzer [11] can detect 7 types of vulnerabilities, *Gasless Send*, *Exception Disorder*, *Reentrancy*, *Timestamp Dependency*, *Block Number Dependency*, *Dangerous DelegateCall* and *Freezing Ether*. SmartCheck [7] has a comprehensive code issue classification for smart contracts in terms of security issues, functional issues, and development issues. The security issues include eight categories of *Balance equality*, *Unchecked external call*, *DoS by external contract*, *send instead of transfer*, *Reentrancy*, *Malicious libraries*, *Using tx.origin*, *Transfer forwards all gas*. In addition, SmartBugs [21] evaluated 9 detection tools on over 47k smart contracts and categorized the vulnerabilities into 10 types, which are *Access Control*, *Bad Randomness*, *Arithmetic*, *Denial of Service*, *Front Running*, *Reentrancy*, *Short addresses*, *Time Manipulation*, *Unchecked Low Level Calls* and *Unknown Unknowns*.

Meanwhile, some literature reviews study a systematic classification of the vulnerabilities of smart contracts. For instance, Zhang et al. [24] collect smart contract bugs from multiple sources and divide these bugs into 9 categories by extending the IEEE Standard Classification for Software Anomalies, namely

- **Data:** Bugs in data definition, initialization, mapping, access, or use, as found in a model, specification, or implementation.
- **Description:** Bugs in the description of the software or its use, installation, or operation.
- **Environment:** Bugs due to errors in the supporting software.
- **Interaction:** Bugs that cause by interaction with other Ethereum addresses.
- **Interface:** Bugs in specification or implementation of an interface.
- **Logic:** Bugs in decision logic, branching, sequencing, or a computational algorithm, as found in natural language specifications or implementation language.
- **Performance:** Bugs that cause increased gas consumption.
- **Security:** Bugs that threaten contract security, such as authentication, privacy/confidentiality, property.
- **Standard:** Nonconformity with a defined standard by taking into account the cause of each bug, the most common form of bugs, and the potential false positives and negatives generated by various detection tools.

Among the above vulnerabilities, *Reentrancy vulnerability* is a well-known vulnerability that caused the infamous DAO [6] attack. We search through over 47k smart contract files and found around 1600 contracts contain more than one *call.value*

keyword, which indicates that it may be overlooked by developers but potential to be exploited by hackers. Therefore, we focus on detecting *reentrancy vulnerability* in this study. Such study subjects are also widely targeted by literature approaches [7], [8], [11], [12], [17], [22], [25]. We will give a concrete examples of our study subject in the following subsection.

B. Vulnerability Example

Reentrancy vulnerability is one of the most dangerous smart contract bugs, which can cause the contract balance (ether) to be stolen by attackers. It happens if the payment function call (i.e., *call.value()*, *deposit.value()*, *transfer()* and so on) in a contract is used to call other contracts while the callee also calls the caller and thus enter the caller again, which will eventually withdraw the entire amount of the caller contract account while the record in the block is only the first withdrawal. For example, in the Listing 1, the attacker calls the *attack* function (line 23) in contract *Attack*, it will execute *withdraw* function in contract *Reentrance* by line 24. When the contract *Reentrance* executes the *withdraw* function (line 3), it will use a *call*-statement to send ether to the contract *Attack* (line 6). In solidity language, however, when an external account or other contract sends ether to a contract address, the fallback function of the callee contract will be executed. Therefore, at this time, the contract *Attack* responds to the transfer using the *Attack.fallback* function (line 27). The *Attack.fallback* function calls the *Reentrance.withdraw* function to withdraw the ether again (line 29). Therefore, contract *Attack* will keep withdrawing the ether from contract *Reentrance* until the gas runs out and statement (*deduct-statement*) deducting the number of tokens held by the contract *Attack* will only be executed once.

```

1 contract Reentrance{
2     mapping(address=>uint) public balance;
3     function withdraw(uint _amount){
4         if(balance[msg.sender] >= _amount) {
5             //reentrancy vulnerability
6             msg.sender.call.value(_amount)();
7
8             //deduct statement
9             balance[msg.sender] -= _amount;
10        }
11    }
12    function() public payable{}
13 }
14
15 -----
16 -----
17
18 contract Attack{
19     Reentrance public entrance;
20     constructor(address _target) public{
21         entrance = Reentrance(_target);
22     }
23     function attack() payable{
24         entrance.withdraw(0.5 ether);
25     }
26     //unnamed function is the fallback function
27     function() public payable{
28         // re-enter the Reentrance contract
29         entrance.withdraw(0.5 ether);
30     }
31 }

```

Listing 1: A Reentrancy Vulnerability Example.

C. Data Flow Graph

Data flow graph (DFG) is a widely used tool for program analysis [26], [27], [28], in which nodes denote program variables and edges denote the dependency relations among these variables. Unlike AST, data flow is same under different abstract grammars for the same source code. Such code structure provides crucial code semantic information for code understanding. Taking *result = max_value - min_value* as an example, programmers do not always follow the naming conventions so that it is hard to understand the semantic of the variable (*result*). Data flow provides a new perspective to understand the semantic of the variable *result*, i.e., the value of *result* comes from *max_value* and *min_value*.

Moreover, in order to help the model learn more general characteristics of vulnerabilities, in this work, we propose the concept of crucial data flow graph (CDFG), a subgraph of DFG, which contains critical information that may trigger vulnerabilities. For instance, *call.value* is the critical information for *reentrancy vulnerability*. We define *crucial nodes* as variables which are at the same line with critical information and which have a direct data flow relation to another *crucial node*. *Crucial nodes* constitute the nodes of the CDFG, and the data flow relationship between them constitutes the edges of the graph.

D. Pre-trained model

Pre-trained models are first pre-trained on a large unsupervised text corpus, and then fine-tuned on downstream tasks. Pre-trained models such as ELMo [29], GPT [30] and BERT [31] have led to strong improvement on numerous natural language processing (NLP) tasks. The success of pre-trained models in NLP also promotes the development of pre-trained models for programming language. Existing works [32], [33], [34], [35], [36] regard a source code as a sequence of tokens and pre-train models on source code to support code-related tasks such as code search, code completion, code summarization, etc. GraphCodeBERT is a pre-trained model for programming language that considers the inherent structure of code, data flow graph (DFG), instead of taking syntactic-level structure of code like abstract syntax tree (AST). It is trained on the CodeSearchNet dataset [37], which includes 2.3M functions of six programming languages paired with natural language documents. It shows better capability and performance compared to other pre-trained models on four downstream tasks: natural language code search, clone detection, code translation, and code refinement.

Therefore, we use GraphCodeBERT [26] as the model backbone and modify it to be able to perform defect detection tasks. To the best of our knowledge, we are the first to apply pre-training techniques to defect detection in smart contracts. The details will be presented in Section III.

E. Motivation

In this section, we use a real-world bug example to show the limitations of using DFG and also the potential benefits that we can obtain via using CDFG. The top left of Figure 1

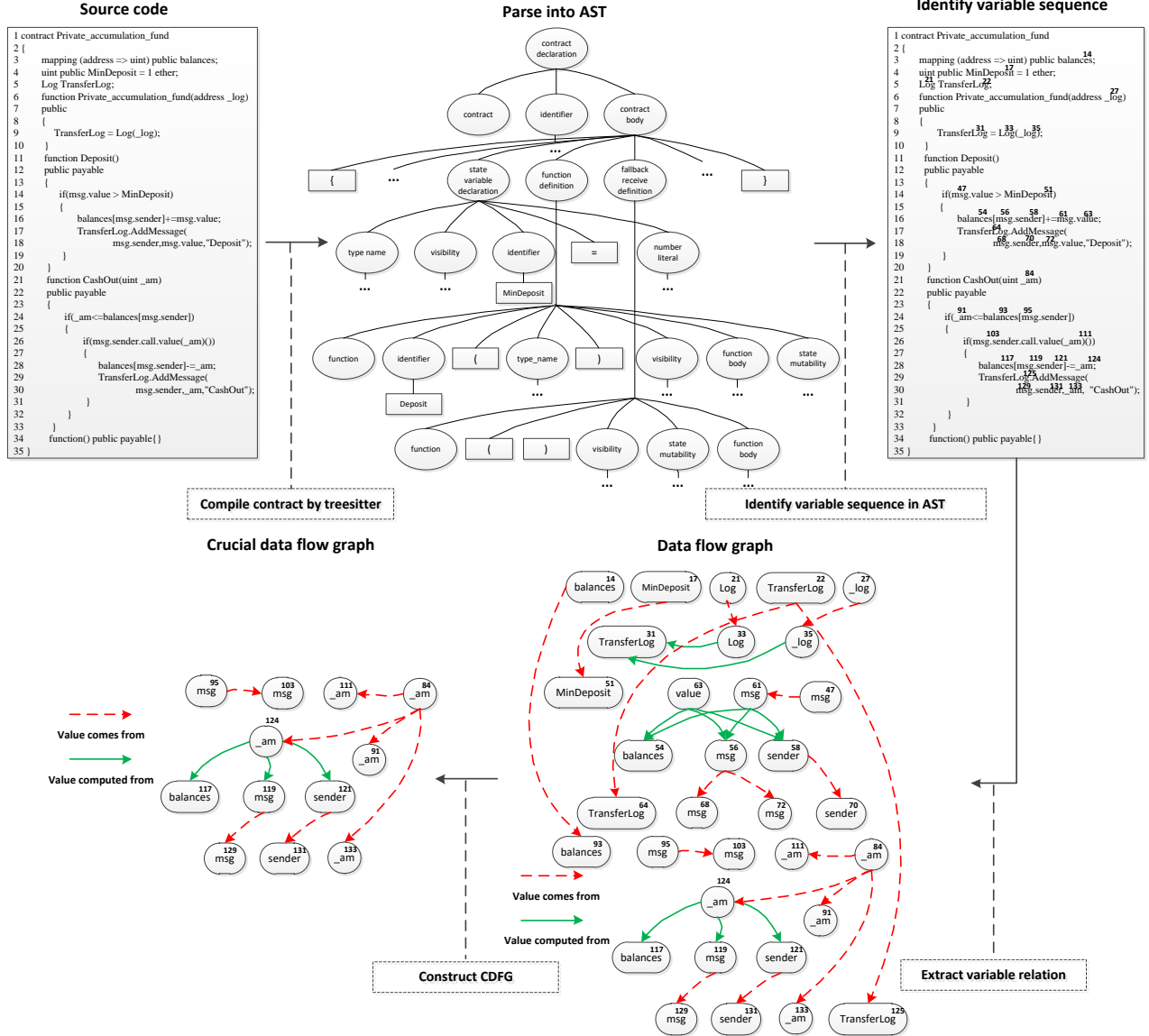


Fig. 1: The procedure of extracting crucial data flow given a source code.

shows a real smart contract deployed on Ethereum, which has a *reentrancy vulnerability* located in line 26-28. In our evaluation (cf. Section IV), our approach successfully detects this vulnerability while the existing approach [17] fails. The DFG and CDFG of this example are shown in the bottom of Figure 1. Here, we argue that in a smart contract, a vulnerability exists in several critical statements, not in all of the contract. Therefore, we can use only the critical information to determine whether a contract is vulnerable or not. In other words, we need to reduce the interference of useless information during vulnerabilities detection.

As mentioned in II-B, the reason why *reentrancy vulnerability* exists is that the atomic transfer operation becomes non-atomic by transferring money first and then deducting the balance, which gives the hacker the opportunity to reentry so

that multiple transfers actually occur but only one deduction of the balance is recorded. Hence, a basic idea to detect *reentrancy vulnerability* is to check whether there is an operation on the balance of the corresponding account after the external call (i.e., call.value()).

Consider the example given in Figure 1. The *reentrancy vulnerability* exists because in function *CashOut* balance operation (in line 28) is after practice transfer (in line 26) with external call call.value(). Other functions such as function *Private_accumulation_fund* and function *Deposit* neither contain payment function call nor have any data interaction with the function *CashOut*. As a result, they are of limited usefulness to help detect the *reentrancy vulnerability*. Therefore, taking into consideration the whole DFG in Figure 1, which contains the data flow information for all functions including

Private_accumulation_fund and *Deposit*, will include some unnecessary information and thus make the graph far more complex comparing with the corresponding concise CDFG. Specifically, DFG contains totally 33 nodes and 28 edges while the number of CDFG are only 12 and 10, with 63.7% and 64.3% reduction respectively.

Moreover, even in the vulnerability related function *CashOut*, the data flow associated with variable *TransferLog* does not affect the occurrence of the *reentrancy vulnerability* either. So we further eliminates the irrelevant data flow information, e.g., *TransferLog* related data flow, to only retain the crucial information related to the vulnerabilities by using CDFG rather than DFG.

The more complex the involved information is, the more difficult it is for the model to train [26], [38]. By reducing the data flow relationships with functions that are not correlated with vulnerabilities in contracts as well as variables that are not correlated with vulnerabilities in related functions, the CDFG helps the model to learn the detection patterns better than DFG. In summary, we decided to use CDFG, which is less complex and does not bring an unnecessarily deep hierarchy, instead of DFG, as the input of our model.

III. OUR APPROACH

Approach overview. The workflow of our approach consists of two phases: (1) a graph generation phase, which extracts the DFG and CDFG from the AST converted from source code, and (2) vulnerability detection phase, which performs smart contract vulnerability detection based on the pre-trained model. Next, we will introduce the two phases, respectively.

A. Graph Generation

Existing work [27] has shown that programs can be transformed into graph representations that preserve the semantic relationships between program elements. Zhuang *et al.* [17] customize smart contracts as contract graphs and set three categories of nodes depending on the importance of program elements in the function. It is desirable to exploit the information of graphs. However, the complex information structure is not conducive to model learning [26].

Our first insight is that key graph information helps the model focus on vulnerabilities more accurately. Therefore, we first parse the source code into an AST, then extract data flow relationships from the AST, and finally transform them into crucial data flow graph (CDFG) according to critical information, i.e., *call.value*. Fig. 1 shows the procedure of extracting crucial data flow given a source code.

Parse into AST. Given a source code $C = \{c_1, c_2, \dots, c_n\}$, we parse the code into AST by a standard compiler tool: tree-sitter [39]. Since tree-sitter does not officially support the Solidity [40] language, which is the most common language used to write smart contracts, we follow JoranHonig’s grammar [41] for solidity, and then modify it to facilitate data flow graph (DFG) generation. The AST includes syntax information of the code and terminals (leaves) are used to identify the variable sequence, denoted as $V = \{v_1, v_2, \dots, v_k\}$.

Algorithm 1: algorithm DFG2CDFG

Input: input $V, E, \text{key_info}, \text{Code}$
Output: output V', E'

```

1  $V' = \emptyset;$ 
2  $E' = \emptyset;$ 
3 for  $v$  in  $V$  do
4   if  $v$  is in the same line with  $\text{key\_info}$  then
5      $\text{add } v$  into  $V'$ ;
6 while  $V'$  becomes larger do
7   for  $v$  in  $V$  do
8     for  $v'$  in  $V'$  do
9       if  $\langle v, v' \rangle$  in  $E$  or  $\langle v', v \rangle$  in  $E$  then
10         $\text{add } v$  into  $V'$ ;
11 for  $\epsilon = \langle v_i, v_j \rangle$  in  $E$  do
12   if  $v_i$  in  $V'$  or  $v_j$  in  $V'$  then
13      $\text{add } \epsilon$  in to  $E'$ 
14 for  $v_s$  in  $V'$  do
15   if none of the edges in  $E'$  pass through the  $v_s$  then
16      $\text{remove } v_s$  from  $V'$ 
17 Return  $V', E'$ 

```

In Fig. 1, for instance, the variable *MinDeposit* in line 4 is ranked 17th in the sequence of variables while in the line 14, the variable *MinDeposit* is the 51st in the variable sequence. They have the same name *MinDeposit* in a data flow but are different variables.

Generate DFG. We take each variable as a node of the graph and a direct edge $\epsilon = \langle v_i, v_j \rangle$ from v_i to v_j indicates that the value of j -th variable comes from or is computed from i -th variable. Taking $x = \text{expr}$ as an example, edges from all variables in expr to x are added into the graph and labeled as “computed from”. Meanwhile, edges like from 17th *MinDeposit* to 51th *MinDeposit* are also added into the graph and labeled as “comes from”. We denote the set of directed edges as $E = \{\epsilon_1, \epsilon_2, \dots, \epsilon_l\}$ and the graph $G(C) = (V, E)$ is data flow used to represent dependency relation between variables of the source code C .

Convert to CDFG. CDFG is the subgraph of DFG, which contains key information that may trigger vulnerabilities. Based on the generated DFG, We retain the data flow associated with critical information by convert DFG to CDFG.

As Algorithm 1 shows, we define *crucial nodes* as variables $V' = \{v'_1, v'_2, \dots, v'_m\}$ which are at the same line with critical information and which have a direct data flow relation to a crucial node, where payment function call (i.e., *call.value()*) is the critical information for *reentrancy vulnerability*. We take each variable in V' as a node of the CDFG and a direct edge $\epsilon' = \langle v'_i, v'_j \rangle$ from v'_i to v'_j which exists if there also exists a same edge in DFG. Moreover, We remove the discrete points in V' through which none of the edges pass. We denote the set of directed edges as $E' = \{\epsilon'_1, \epsilon'_2, \dots, \epsilon'_n\}$ and the graph $CG(C) = (V', E')$ is crucial data flow used to

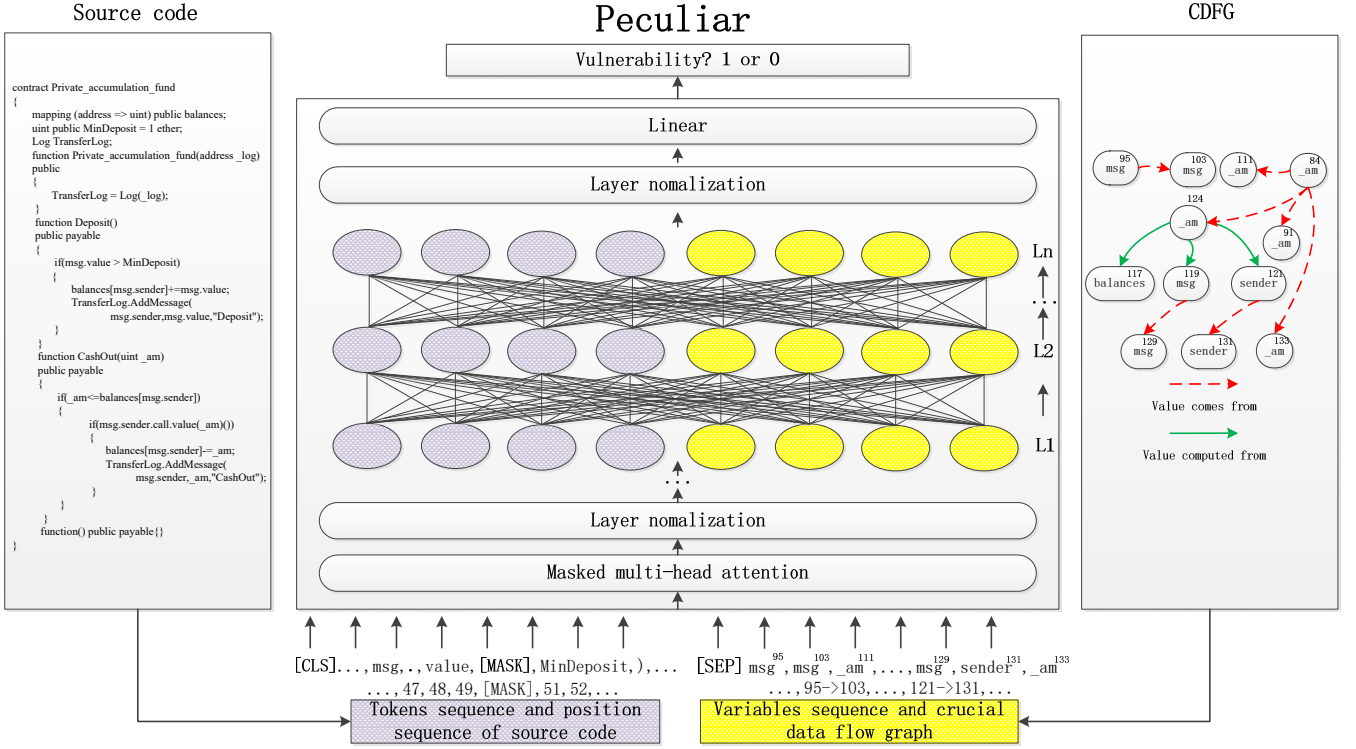


Fig. 2: Peculiar's Architecture.

represent crucial dependency relation between crucial nodes of the source code C .

For example, msg_{103} , $send_{105}$ and am_{111} are the *crucial node* in Figure 1, and then the variables on the data flow path through the existing crucial node are also expanded as crucial nodes. When the number of crucial node is no longer increasing, CDFG is successfully built.

B. Our Model

Our second insight is that the potential of pre-trained model may offer a new perspective probably benefiting vulnerability detection in smart contract. So we customize a graph-based pre-trained model for vulnerability detection. In this section, we describe Peculiar, a detection model based on GraphCodeBert [26], which is a graph-based pre-trained model. We introduce model architecture, graph-guided masked attention and training tasks in the following.

Model Architecture. We can see the architecture of our model in Fig. 2. We follow GraphCodeBERT [26], which is based on Transformer neural architecture [38] for programming language, and use a Linear layer to output the result.

Unlike GraphCodeBERT where the input contains data flow graph (DFG), we convert the DFG to crucial data flow graph (CDFG) and take it as the input for our model. Specifically, from the source code $SC = \{sc_1, sc_2, \dots, sc_n\}$ we first obtain the corresponding CDFG (as discussed in Section III-A) $CG(SC) = (Var, Edge)$, where $Var = \{v_1, v_2, \dots, v_k\}$ is a variable set, $Edge = \{\epsilon_1, \epsilon_2, \dots, \epsilon_l\}$ is a directed edge set indicating where the value of each variable comes from. We then concatenate the source code and variable set into a

sequence $I = \{[CLS], SC, [SEP], Var\}$ to be input into the model, where $[CLS]$ is a special token in front of the two sets and $[SEP]$ is a special notation to split the source code SC and the variable set Var .

After being put into Peculiar, the sequence I is transformed into the input vector X^0 . Specifically, for each token in I , we not only embed itself but also generate embedding for its position and add these two embeddings to represent this token; for all variables in I , we use special position embeddings (i.e., $95 \rightarrow 103$ as shown in the CDFG of Fig. 2) to show the data flow relationships.

The input vector X^0 goes through $N=12$ transformer layers in Peculiar to generate contextual representations, $X^n = \text{transformer}_n(X^{n-1})$, $n \in [1, N]$, where each transformer layer contains a structurally equivalent transformer and the vector X^{n-1} will first generate the vector H^n after a multi-headed self-attentive operation [38], and then output the vector X^n after a feed-forward layer.

$$H^n = \text{LN}(\text{MHSA}(X^{n-1}) + X^{n-1}) \quad (1)$$

$$X^n = \text{LN}(\text{FFN}(H^n) + H^n) \quad (2)$$

where MHSA is a multi-headed self-attention mechanism, FFN is a two layers feed forward network, and LN represents a layer normalization operation. For the n -th transformer layer, the output \hat{X}^n of a multi-headed self-attention is computed as:

$$Q_i = X^{n-1}W_i^Q, K_i = X^{n-1}W_i^K, V_i = X^{n-1}W_i^V \quad (3)$$

$$head_i = \text{softmax}(\frac{Q_i K_i^T}{\sqrt{d_k}} + M) V_i \quad (4)$$

$$\hat{X}^n = [head_1; \dots; head_m] W_n^O \quad (5)$$

where the previous layer's output $X^{n-1} \in \mathbb{R}^{|I| \times d_h}$ is linearly projected onto a triplet consisting of queries, keys, and values using model parameters $W_i^Q, W_i^K, W_i^V \in \mathbb{R}^{d_h \times d_k}$, respectively. m is the number of heads, d_k is the dimension of a head, and $W_n^O \in \mathbb{R}^{d_h \times d_h}$ is the model parameters. $M \in \mathbb{R}^{|I| \times |I|}$ is a mask matrix, where M_{ij} is 0 if i -th token is allowed to attend j -th token otherwise $-\infty$.

At the end of the model, we add a linear classifier and use the *Sigmoid* function to output the predicted probabilities \hat{y}

$$\hat{y} = \text{Sigmoid}(\hat{X}^n) \quad (6)$$

Graph Masked Attention. In order to introduce graph structure into Transformer and represent dependency relation between variables, we use graph-guided masked attention function to model token relations, following the GraphCodeBERT [26].

We use mask matrix M to demonstrate graph-guided masked attention in the equation 7:

$$M_{ij} = \begin{cases} 0 & \text{if } query_i \in \{[CLS], [SEP]\} \\ & \text{or } query_i, key_j \in SC \\ & \text{or } \langle query_i, key_j \rangle \in Edge \cup Edge' \\ -\infty & \text{otherwise} \end{cases} \quad (7)$$

where $query_{var_i}$ means query of node var_i , key_{var_i} is the node-key of node var_i . $Edge'$ is a set to represent the relation between source code tokens and nodes of the data flow, where $\langle var_i, code_j \rangle / \langle code_j, var_i \rangle \in Edge'$ if the variable var_i is identified from the source code token $code_j$.

If there is a direct edge between node var_i and node var_j (i.e., $\langle var_i, var_j \rangle \in Edge$) or they are the same node (i.e., $i = j$), then the node query $query_{var_j}$ is allowed to pay attention to a node-key key_{var_i} . Otherwise, the attention is masked, adding the attention score to an infinitely large negative value. By adding an infinite negative value to the attention score $query_j^T key_i$, the attention weight becomes zero after using the *softmax* function (i.e., in equation 4). Then the attention masking function could avoid querying the key key_i attended by the query $query_j$.

In addition, we allow the node $query_{var_i}$ and code key_{code_j} attend each other if and only if $\langle var_i, code_j \rangle / \langle code_j, var_i \rangle \in Edge'$.

C. Training Tasks

We next describe two training tasks, namely the pre-training task, which contains three subtasks, and the detection task. **Pre-training tasks.** The first pre-training task is masked language modeling [31], which aims to learn representation from the source code. The second pre-training task is data flow edge prediction for learning representation from data flow. The last pre-training task is variable-alignment across source

code and data flow, which is used to align the representation between source code and data flow and to predicts where a variable is identified from. All these pre-training tasks follow those from GraphCodeBERT [26].

Detection task. This task aims to find out potential vulnerabilities exist in smart contracts by fine-tuning the model. During training, networks are fed with a large number of smart contract source code and corresponding CDFG, together with their ground truth labels. Then, the trained models are employed to absorb a pair with source code and CDFG and yield a vulnerability detection label. It is also important to mention that we have developed automation tools for converting source code to CDFG, so that the whole process is fully automated.

IV. EXPERIMENTS

To evaluate our approach, we design experiments to answer the following research questions:

- **RQ1:** *How effective is Peculiar at detecting vulnerabilities in smart contracts?* We first aim to understand how well Peculiar performs on smart contract vulnerabilities detection compared to other state-of-the-art approaches. Specifically, we concentrate on Recall, Precision and F1 when detecting *reentrancy vulnerability*.
- **RQ2:** *Does Peculiar detect reentrancy vulnerability only based on keywords that are related with reentrancy vulnerability?* We note that most *reentrancy vulnerability* come up with the keywords about external function. However, it should be noted that not all contracts with these keywords are vulnerable. Likely, not all reentrancy vulnerabilities contain these keywords. Therefore, it is worthy to investigate whether our Peculiar is overfitting to these keywords. We are thus motivated to perform this experiment where we evaluate the performance of Peculiar only on contracts with these keywords.
- **RQ3:** *How the different modules contribute to the Peculiar?* We try to investigate how the different modules contribute to the Peculiar, including CDFG and pre-trained model. We design ablation experiments to investigate this research question.

A. Dataset and Experimental Settings

Dataset. For RQ1 and RQ3, we use the **SmartBugs Wild Dataset** [21], which is a recently-released, large-scale, and Solidity language based dataset for smart contract vulnerabilities, as our benchmark. We call this dataset as *dataset-wild*, which contains 47,398 real and unique sol files with roughly 203,716 contracts in total (One .sol file contains one or more contracts). We manually labeled each of these smart contract for model learning. The first two authors labeled them individually and they finally reached the consensus after a discussion. The labeled dataset has been open sourced in our online repository. For RQ2, we select all contracts containing the keywords which may cause *reentrancy vulnerability* from the *dataset-wild* as the *dataset-vul*, which contains 1,197 vulnerable contracts and 471 non-vulnerable contracts.

TABLE I: Performance comparison of the involved approaches in terms of Recall, Precision and F1 score on the *dataset-wild*.

Method	Reentrancy		
	Recall(%)	Precision(%)	F1(%)
Honeybadger	50.54	87.21	50.92
Manticore	49.99	49.70	49.85
Mythril	51.69	50.24	49.74
Osiris	53.82	59.01	55.33
Oyente	54.11	65.63	56.44
Securify	54.81	52.63	53.36
Slither	65.41	51.97	52.60
Smartcheck	70.50	79.37	74.14
DR-GCN	80.89	72.36	76.39
TMP	82.63	74.06	78.11
Peculiar	92.40	91.80	92.10

Experimental settings. For RQ1, We compare our approach with a total of ten state-of-the-art approaches on *dataset-wild*, namely Honeybadger [42], Manticore [15], Mythril [14], Osiris [13], Oyente [12], Securify [10], Slither [8], SmartCheck [7] and two approaches based machine learning (DR-GCN [17] and TMP [17]). The involved baseline approaches are representatives of a wide range of detection tool categories (e.g., deep learning based and program analysis based). For RQ2, We use Peculiar to compare with eight aforementioned approaches excluding DR-GCN [17] and TMP [17] on the *dataset-vul*. The authors do not provide source code or detailed result for each individual contract so that we can neither calculate the performances of these two tools nor reproduce their experiments. For RQ3, first we respectively use CDFG and DFG as input for Peculiar and conducted ablation experiments on *dataset-wild* to analyze the contribution of CDFG. Then, we normalize the parameters of the pre-trained model and retrain it with the input of CDFG to investigate the contribution of pre-trained model. The experimental environment is set up with Ubuntu18.04 system, 64G RAM, i7-9700 CPU and NVIDIA 1080ti graphics card.

As for the evaluation metrics, we adopt the widely used *Precision*, *Recall*, and *F1-score* [7], [17]. We choose the *macro* way to perform the evaluation which will calculate values with respect to the three metrics for contracts with and without vulnerabilities, respectively, and then take the average value as the final result. Such a way can reflect the general performance of our approach.

For each research question, we randomly pick 20% contracts as the training set, and the left part is served as the test set, following Zhuang *et al.* [17].

B. Experimental Results

We now demonstrate the experimental results to answer the proposed research questions in this paper.


1) *RQ1: [Effectiveness of Detecting Vulnerabilities]:* We compare our Peculiar with the state-of-the-art smart contract vulnerability detection tools, namely Honeybadger [42], Manticore [15], Mythril [14], Osiris [13], Oyente [12], slither [8], Smartcheck [7], Securify [10], DR-GCN [17] and TMP

TABLE II: Discernment comparison of different tools to *reentrancy vulnerability* in terms of Recall, Precision and F1 score on the *dataset-vul*.

Method	Reentrancy		
	Recall*(%)	Precision*(%)	F1*(%)
Honeybadger	50.54	64.70	23.79
Manticore	49.85	14.55	22.53
Mythril	44.07	37.43	26.38
Osiris	46.30	41.51	27.62
Oyente	47.28	43.70	28.40
Securify	52.09	55.06	32.17
Slither	60.95	60.99	50.30
Smartcheck	54.17	53.63	48.28
Peculiar	82.61	84.09	83.29

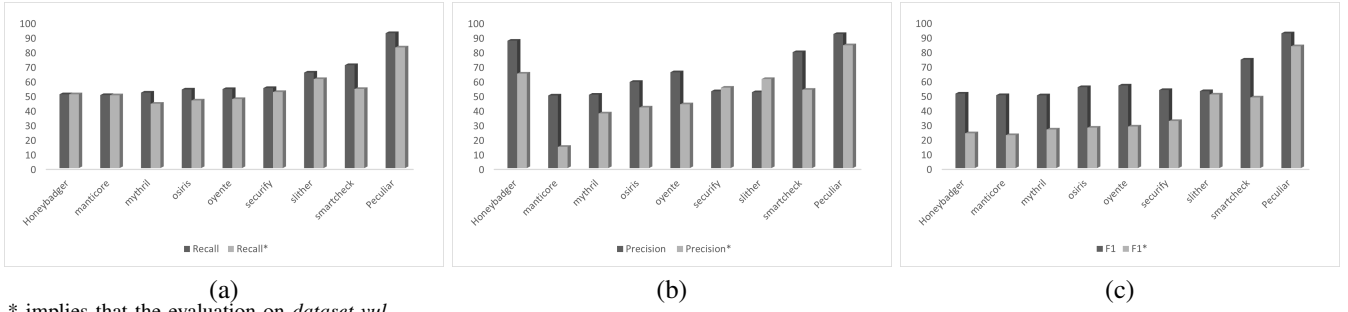
[17] on the *dataset-wild*. The performances of different approaches are presented in the Table I.

From the quantitative results of Table I, we have the following observations. First, we found that the existing tools have not yet achieved satisfactory precision in *reentrancy vulnerability* detection, e.g., among baseline approaches, the highest precision is 87.21% while the average precision is only 64.22%. Second, Peculiar outperforms existing approaches to a large extent. More specifically, Peculiar achieves a precision of 91.80%, improving the state-of-the-art by 5.3%. With respect to the recall, it outperforms the state-of-the-art by 11.8%. In addition, the F1 score of Peculiar is 17.91% higher than the maximum of the existing techniques, showing that Peculiar achieves a significant improvement with respect to the overall performance.

RQ-1  Peculiar outperforms the existing approaches in detecting smart contracts vulnerabilities. For instance, its F1 score is 17.91% higher than that from the state-of-the-art.

2) *RQ2: [Discernment to Reentrancy vulnerability]:* Meanwhile, to better observe the discernment of different tools to *reentrancy vulnerability*, we compare the performances of different approaches on the *dataset-vul* dataset. The results are shown in Table II where the metrics recall, precision and F1 score are marked by *. We also plot the performance of the different tools for the three metrics on the two datasets (i.e., *dataset-wild* and *dataset-vul*) in Figure 3 to visualize the change.

According to the results shown in Table I, Table II and Figure 3, we find that there is a large space for improvement for the discrimination ability of existing approaches on *reentrancy vulnerability*. For instance, Manticore [15] achieves a precision of only 14.55% on *dataset-vul* while the F1 score of Smartcheck [7] drops by 34.88%, from 74.14% on *dataset-wild* to 48.28% on *dataset-vul*. The mean values of these baseline approaches in terms of precision, recall and F1 score are 46.45%, 50.66% and 32.43%, respectively. In comparison, Peculiar performs well in the identification of *reentrancy vulnerability*, with precision of 84.09%, recall of 82.61% and F1 of 83.29% on the *dataset-vul*.



* implies that the evaluation on *dataset-vul*.

Fig. 3: Results of the comparison of three metrics for Peculiar and other approaches on *dataset-wild* and *dataset-vul*.

TABLE III: Results of the ablation study.

Metrics	Reentrancy	
	Peculiar	Peculiar-WOS
Recall(%)	92.40	68.26
Precision(%)	91.80	88.16
F1(%)	92.10	74.66

RQ-2 *Peculiar has higher discernment to reentrancy vulnerability. Specifically, its precision, recall and F1 score are 29.96%, 35.53% and 65.58% higher than the maximums of the existing approaches respectively.*

3) RQ3: [Contribution of CDFG and pre-trained model]:

By default, Peculiar adopts the crucial data flow graph to highlight the crucial information in the graph. It is thus interesting to see the contribution of CDFG in our model, in other words, what is the effect of using DFG instead of CDFG. We removed the *Purification operations* from Peculiar and compared the result with that of the default Peculiar. The variant is denoted as Peculiar-WOS where WOS is short for *without streamlining*. Quantitative results are summarized in Table III where we can see that without the streamlining step, the performance of Peculiar decreases significantly. For example, the Peculiar witnesses a 35.36% and 23.35% decrease in terms of Recall and F1 score, respectively.

Moreover, in order to see the contribution of the pre-training technique we utilized, we normalize the parameters of the pre-trained model and retrain it with the input of CDFG. During the training process, the loss rate of the model decreased insignificantly and the model did not converge. We then experimented with the model on *dataset-wild* after a period of training and found that the model did not find any vulnerable contract, which is in contrast to the pre-trained model. This result illustrates that pre-training technique plays an irreplaceable role in vulnerabilities detection by improving the generality as well as reducing the learning efforts of the model.

RQ-3 *CDFG combines better than DFG with our pre-trained model, and the pre-trained model plays an irreplaceable role in Peculiar.*

V. DISCUSSION

A. Data Flow Graph in this Study

From the perspective of program analysis, a valid data flow graph needs to be generated by examining control flow graph

(CFG) and calculating the *gen* and *kill* sets. However, in our paper, we focus more on representing the data dependency relations among variables through data dependency graph (a variant of data flow graph [43]), and obtaining the dataflow facts of the program through the *worklist* algorithm is not our focus. Therefore, we choose to generate data flow graphs based on ASTs via following the existing study [17].

B. Threats to Validity

The threats to validity of this study mainly come from two aspects. First, the ground truth in our dataset (i.e., whether a contract is vulnerable or not) is manually labelled. Such a process may suffer from human subjectivity [44], [45] and studies where the ground truth is manually defined [46], [47], [48] are widely affected by this factor. Nevertheless, this threat is mitigated considering that two authors independently labelled the dataset and an agreement was finally achieved when the initial result is controversial. Moreover, the dataset is open sourced for further reviews.

Second, Peculiar currently has only been evaluated on detecting *reentrancy vulnerability*. The reason for this limitation is from the off-the-shelf dataset. So far, there is no recognized large-scale publicly-available dataset on smart contract vulnerabilities. Therefore, we manually labelled the *reentrancy vulnerability* to support our research. Evaluating the performances of Peculiar on other vulnerability types requires another large-scale manual labelling process which is rather time-consuming and thus is left as our future work. This threat is also mitigated considering that *reentrancy vulnerability* is one of the most severe and common smart contract vulnerabilities (cf. Section II) and existing studies also explicitly focus on this type of vulnerability (e.g., [22]).

VI. RELATED WORK

A. Smart contract vulnerability detection

Program Analysis. Program analysis is a general computer technology that aims at obtaining program characteristics and properties by automating the analysis process of programs. Representative tools or frameworks for smart contracts include SmartCheck [7], SASC [49], Slither [8], etc.

Formal Validation. Formal verification techniques are effective ways for verifying that a program conforms to the expected design properties and security specifications. Representative tools or frameworks for smart contracts are ZEUS [9], Securify [10], VerX [50], etc.

Fuzzy Test. Fuzzy testing is a powerful software analysis technique. The core idea is to provide a large number of test cases for a program to monitor its abnormal behavior during execution in order to find program vulnerabilities. Representative tools or frameworks for smart contracts are Echidna [51], ContractFuzzer [11], ILF [52], Harvey [53], etc.

Machine Learning. SmartEmbed [16] determines the presence of vulnerabilities by calculating the similarity to smart contracts with known bugs based on deep learning model. S-gram [54] uses Oyente [12] to obtain ground truth and combines N-gram language model and lightweight static semantic tagging to predict potential vulnerabilities. Tann *et al.* [55] use MAIAN to label security issues and leverage LSTM to predict potential flaws. Huang *et al.* [56] first convert bytecode to RGB colors and then use convolutional neural networks to train models and predict security issues based on manually labeled datasets. Zhuang *et al.* [17] use graph neural networks for defect detection in smart contracts, which can detect *reentrancy vulnerability*, *timestamp dependence vulnerability*, and *infinite loop vulnerability*.

Symbolic Execution. Symbolic execution is a traditional automated vulnerability mining technique, which is now also widely used for smart contract vulnerability mining. Representative tools or frameworks for smart contracts include Oyente [12], Osiris [13], Mythril [14], Manticore [15], etc.

Taint Analysis. Taint analysis is a special kind of program analysis technique that enables accurate program analysis by marking critical data of interest and tracking its flow during program execution. The most representative tool for smart contracts is Sereum [57]. Furthermore, some aforementioned approaches also utilize the results of taint analysis in their workflows such as Oyente [12] and Mythril [14].

In addition, there is a recent work [22] detecting practical reentrancy vulnerabilities. It proposed *Clairvoyance*, a cross-function and cross-contract static analysis approach to detect reentrancy vulnerabilities in real world, using five major path protective techniques (PPTs) to support fast yet precise path feasibility checking. *Clairvoyance* has achieved pretty good performance in resolving false negatives and false positives.

B. Deep Learning in Software Engineering

Pre-Trained Models for Programming Languages. Inspired by the big success of pre-training techniques in NLP [31], [58], [59], [60], pre-trained models for programming languages also promote the development of software engineering tasks [32], [34], [33], [35], [36], [32]. Feng *et al.* [34] propose CodeBERT, a bimodal pre-trained model for programming and natural languages by masked language modeling and replaced token detection to support text-code tasks such as code search. Karampatsis and Sutton [33] pre-train contextual embeddings on a JavaScript corpus using the ELMo framework for program repair task. Svyatkovskiy *et al.* [35] propose GPT-C, which is a variant of the GPT-2 trained from scratch on source code data to support generative tasks like code completion. Buratti *et al.* [36] present C-BERT, a transformer-based language model pre-trained on a collection of repositories written in C

language, and achieve high accuracy in the abstract syntax tree (AST) tagging task. Guo *et al.* [26] propose GraphCodeBERT, the first pre-trained model that leverages code structure to learn code representation to improve code understanding.

Program Analysis Techniques Based on Program Structure. Zhang *et al.* [61] propose an AST-based Neural Network for source code representation and apply it to two common program comprehension tasks: source code classification and code clone detection. DeepBugs [62] represents code via word2vec for detecting name-based bugs. ADF-GA [63], All-uses Data Flow criterion based test case generation using Genetic Algorithm, is a novel test case generation approach for dynamic testing of smart contract programs. Allamanis *et al.* [27] performs Gated Graph Neural Networks on program graphs which track the dependencies of the same variables and functions to predict variable names.

Other Deep Learning Based Techniques in Software Engineering. In recent years, there are also many emerging deep learning applications in the software engineering fields. CNN-FL [64] uses Convolutional Neural Networks to help localizing faults; Li *et al.* [65] propose DeepFL which has good performance on fault localization based on deep learning; Zhang *et al.* [66] introduce the effectiveness of deep learning in locating real faults; DeepAPI [67] uses a sequence-to-sequence (seq2seq) neural network to learn representations of natural language queries and predict relevant API sequences; Cognac [68] leverages a seq2seq model optimized by the prior knowledge which is summarized from a large-scale dataset to recommend high quality method names; Lam *et al.* [69] combines deep neural network with information retrieval (IR) technique to recommend potential buggy files; and a joint embedding model is used in code search to map source code and natural language descriptions into a unified vector space for evaluating semantics similarity [70].

VII. CONCLUSION

In this paper, we propose an automated approach *Peculiar* to detect *reentrancy vulnerability* in smart contracts. To the best of our knowledge, *Peculiar* is the first smart contract vulnerability detection approach that is based on pre-training techniques and the crucial data flow graph of smart contracts. Compared to existing approaches, the extracted crucial data flow graphs in contracts not only consider value dependencies between program variables and functions but also focus on the critical information related to vulnerabilities. In addition, we also explore the feasibility of using pre-trained models for vulnerability detection. Extensive experiments show that our approach significantly outperforms the state-of-the-art approaches and other neural networks. Our work is an important step in revealing the potentiality of pre-trained model approaches for smart contract vulnerability detection tasks.

All source code and data in this study are publicly available now at:

<https://github.com/wuhongjun15/Peculiar>.

REFERENCES

- [1] W. Chen, M. Ma, Y. Ye, Z. Zheng, and Y. Zhou, "Iot service based on jointcloud blockchain: The case study of smart traveling," in *2018 IEEE Symposium on service-oriented system engineering (SOSE)*. IEEE, 2018, pp. 216–221.
- [2] Ethereum, "Erc20," <https://github.com/ethereum/EIPs/blob/master/EIPS/eip-20.md>.
- [3] D. L. Inc., "Cryptokitties," <https://www.cryptokitties.co/>.
- [4] Y. Velner, J. Teutsch, and L. Luu, "Smart contracts make bitcoin mining pools vulnerable," in *International Conference on Financial Cryptography and Data Security*. Springer, 2017, pp. 298–316.
- [5] P. Technologies, "parity," <https://paritytech.io/security-alert-2/>.
- [6] D. Siegel, "Understanding the dao attack," <https://www.coindesk.com/understanding-dao-hack-journalists>.
- [7] S. Tikhomirov, E. Voskresenskaya, I. Ivanitskiy, R. Takhaviev, E. Marchenko, and Y. Alexandrov, "Smartcheck: Static analysis of ethereum smart contracts," in *Proceedings of the 1st International Workshop on Emerging Trends in Software Engineering for Blockchain*, 2018, pp. 9–16.
- [8] J. Feist, G. Grieco, and A. Groce, "Slither: a static analysis framework for smart contracts," in *2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*. IEEE, 2019, pp. 8–15.
- [9] S. Kalra, S. Goel, M. Dhawan, and S. Sharma, "Zeus: Analyzing safety of smart contracts," in *Ndss*, 2018, pp. 1–12.
- [10] P. Tsankov, A. Dan, D. Drachsler-Cohen, A. Gervais, F. Buenzli, and M. Vechev, "Securify: Practical security analysis of smart contracts," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 67–82.
- [11] B. Jiang, Y. Liu, and W. Chan, "Contractfuzzer: Fuzzing smart contracts for vulnerability detection," in *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2018, pp. 259–269.
- [12] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making smart contracts smarter," in *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, 2016, pp. 254–269.
- [13] C. F. Torres, J. Schütte, and R. State, "Osiris: Hunting for integer bugs in ethereum smart contracts," in *Proceedings of the 34th Annual Computer Security Applications Conference*, 2018, pp. 664–676.
- [14] B. Mueller, "Mythril-reversing and bug hunting framework for the ethereum blockchain," 2017.
- [15] M. Mossberg, F. Manzano, E. Hennenfent, A. Groce, G. Grieco, J. Feist, T. Brunson, and A. Dinaburg, "Manticore: A user-friendly symbolic execution framework for binaries and smart contracts," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 1186–1189.
- [16] Z. Gao, V. Jayasundara, L. Jiang, X. Xia, D. Lo, and J. Grundy, "Smartembed: A tool for clone and bug detection in smart contracts through structural code embedding," in *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2019, pp. 394–397.
- [17] Y. Zhuang, Z. Liu, P. Qian, Q. Liu, and Q. He, "Smart contract vulnerability detection using graph neural network," in *Twenty-Ninth International Joint Conference on Artificial Intelligence and Seventeenth Pacific Rim International Conference on Artificial Intelligence IJCAI-PRICAI-20*, 2020.
- [18] T. Zhang, B. Xu, F. Thung, S. A. Haryono, D. Lo, and L. Jiang, "Sentiment analysis for software engineering: How far can pre-trained transformer models go?" in *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2020, pp. 70–80.
- [19] F. Liu, G. Li, Y. Zhao, and Z. Jin, "Multi-task learning based pre-trained language model for code completion," in *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2020, pp. 473–485.
- [20] R. Robbes and A. Janes, "Leveraging small software engineering data sets with pre-trained neural networks," in *2019 IEEE/ACM 41st International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*, 2019, pp. 29–32.
- [21] J. F. Ferreira, P. Cruz, T. Durieux, and R. Abreu, "Smartbugs: A framework to analyze solidity smart contracts," in *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2020, pp. 1349–1352.
- [22] Y. Xue, M. Ma, Y. Lin, Y. Sui, J. Ye, and T. Peng, "Cross-contract static analysis for detecting practical reentrancy vulnerabilities in smart contracts," in *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2020, pp. 1029–1040.
- [23] N. F. Samreen and M. H. Alalfi, "Reentrancy vulnerability identification in ethereum smart contracts," in *2020 IEEE International Workshop on Blockchain Oriented Software Engineering (IWBOSE)*. IEEE, 2020, pp. 22–29.
- [24] P. Zhang, F. Xiao, and X. Luo, "A framework and dataset for bugs in ethereum smart contracts," in *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2020, pp. 139–150.
- [25] A. Ghaleb and K. Pattabiraman, "How effective are smart contract analysis tools? evaluating smart contract static analysis tools using bug injection," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2020, pp. 415–427.
- [26] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, S. Liu, L. Zhou, N. Duan, J. Yin, D. Jiang *et al.*, "Graphcodebert: Pre-training code representations with data flow," *arXiv preprint arXiv:2009.08366*, 2020.
- [27] M. Allamanis, M. Brockschmidt, and M. Khademi, "Learning to represent programs with graphs," in *International Conference on Learning Representations*, 2018.
- [28] V. J. Hellendoorn, C. Sutton, R. Singh, P. Maniatis, and D. Bieber, "Global relational models of source code," in *International conference on learning representations*, 2019.
- [29] M. Peters, M. Neumann, M. Iyyer, M. Gardner, and L. Zettlemoyer, "Deep contextualized word representations," in *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers)*, 2018.
- [30] A. Radford, K. Narasimhan, T. Salimans, and I. Sutskever, "Improving language understanding by generative pre-training," 2018. [Online]. Available: https://s3-us-west-2.amazonaws.com/openai-assets/research-covers/language-unsupervised/language_understanding_paper.pdf
- [31] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.
- [32] A. Kanade, P. Maniatis, G. Balakrishnan, and K. Shi, "Pre-trained contextual embedding of source code," *arXiv preprint arXiv:2001.00059*, 2019.
- [33] R.-M. Karampatsis and C. Sutton, "Scelmo: Source code embeddings from language models," *arXiv preprint arXiv:2004.13214*, 2020.
- [34] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang *et al.*, "Codebert: A pre-trained model for programming and natural languages," *arXiv preprint arXiv:2002.08155*, 2020.
- [35] A. Svyatkovskiy, S. K. Deng, S. Fu, and N. Sundaresan, "Intellicode compose: Code generation using transformer," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, pp. 1433–1443.
- [36] L. Buratti, S. Pujar, M. Bornea, S. McCarley, Y. Zheng, G. Rossiello, A. Morari, J. Laredo, V. Thost, Y. Zhuang *et al.*, "Exploring software naturalness through neural language models," *arXiv preprint arXiv:2006.12641*, 2020.
- [37] H. Husain, H.-H. Wu, T. Gazit, M. Allamanis, and M. Brockschmidt, "Codesearchnet challenge: Evaluating the state of semantic code search," *arXiv preprint arXiv:1909.09436*, 2019.
- [38] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," in *NIPS*, 2017.
- [39] T. gan LUA *et al.*, "tree-sitter," <https://tree-sitter.github.io/tree-sitter/>.
- [40] C. Dannen, *Introducing Ethereum and solidity*. Springer, 2017, vol. 318.
- [41] JoranHonig, "tree-sitter-solidity," <https://github.com/JoranHonig/tree-sitter-solidity>.
- [42] C. F. Torres, M. Steichen *et al.*, "The art of the scam: Demystifying honeypots in ethereum smart contracts," in *28th {USENIX} Security Symposium ({USENIX} Security 19)*, 2019, pp. 1591–1607.
- [43] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 9, no. 3, pp. 319–349, 1987.
- [44] S. Wang, M. Wen, L. Chen, X. Yi, and X. Mao, "How different is it between machine-generated and developer-provided patches? an empirical study on the correct patches generated by automated program repair techniques," in *Proceedings of the 13th International Symposium*

- on *Empirical Software Engineering and Measurement*. IEEE, 2019, pp. 1–12.
- [45] X.-B. D. Le, L. Bao, D. Lo, X. Xia, S. Li, and C. Pasareanu, “On reliability of patch correctness assessment,” in *Proceedings of the 41st International Conference on Software Engineering*. IEEE, 2019, pp. 524–535.
- [46] K. Liu, S. Wang, A. Koyuncu, K. Kim, T. F. Bissyandé, D. Kim, P. Wu, J. Klein, X. Mao, and Y. L. Traon, “On the efficiency of test suite based program repair: A systematic assessment of 16 automated repair systems for java programs,” in *Proceedings of the 42nd International Conference on Software Engineering*. ACM, 2020, pp. 615–627.
- [47] S. Wang, M. Wen, B. Lin, H. Wu, Y. Qin, D. Zou, X. Mao, and H. Jin, “Automated patch correctness assessment: How far are we?” in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. ACM, 2020.
- [48] Y. Qin, S. Wang, K. Liu, X. Mao, and T. F. Bissyandé, “On the impact of flaky tests in automated program repair,” in *Proceedings of the 28th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2021, pp. 295–306.
- [49] E. Zhou, S. Hua, B. Pi, J. Sun, Y. Nomura, K. Yamashita, and H. Kurihara, “Security assurance for smart contract,” in *2018 9th IFIP International Conference on New Technologies, Mobility and Security (NTMS)*. IEEE, 2018, pp. 1–5.
- [50] A. Permenev, D. Dimitrov, P. Tsankov, D. Drachsler-Cohen, and M. Vechev, “Verx: Safety verification of smart contracts,” in *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2020, pp. 1661–1677.
- [51] G. Grieco, W. Song, A. Cygan, J. Feist, and A. Groce, “Echidna: effective, usable, and fast fuzzing for smart contracts,” in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2020, pp. 557–560.
- [52] 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 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 531–548.
- [53] V. Wüstholz and M. Christakis, “Harvey: A greybox fuzzer for smart contracts,” in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, pp. 1398–1409.
- [54] H. Liu, C. Liu, W. Zhao, Y. Jiang, and J. Sun, “S-gram: towards semantic-aware security auditing for ethereum smart contracts,” in *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2018, pp. 814–819.
- [55] W. J.-W. Tann, X. J. Han, S. S. Gupta, and Y.-S. Ong, “Towards safer smart contracts: A sequence learning approach to detecting security threats,” *arXiv preprint arXiv:1811.06632*, 2018.
- [56] T. H.-D. Huang, “Hunting the ethereum smart contract: Color-inspired inspection of potential attacks,” *arXiv preprint arXiv:1807.01868*, 2018.
- [57] M. Rodler, W. Li, G. O. Karame, and L. Davi, “Sereum: Protecting existing smart contracts against re-entrancy attacks,” *arXiv preprint arXiv:1812.05934*, 2018.
- [58] Z. Yang, Z. Dai, Y. Yang, J. Carbonell, R. Salakhutdinov, and Q. V. Le, “Xlnet: Generalized autoregressive pretraining for language understanding,” *arXiv preprint arXiv:1906.08237*, 2019.
- [59] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov, “Roberta: A robustly optimized bert pretraining approach,” *arXiv preprint arXiv:1907.11692*, 2019.
- [60] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu, “Exploring the limits of transfer learning with a unified text-to-text transformer,” *arXiv preprint arXiv:1910.10683*, 2019.
- [61] J. Zhang, X. Wang, H. Zhang, H. Sun, K. Wang, and X. Liu, “A novel neural source code representation based on abstract syntax tree,” in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 783–794.
- [62] M. Pradel and K. Sen, “Deepbugs: A learning approach to name-based bug detection,” *Proceedings of the ACM on Programming Languages*, vol. 2, no. OOPSLA, pp. 1–25, 2018.
- [63] P. Zhang, J. Yu, and S. Ji, “Adf-ga: Data flow criterion based test case generation for ethereum smart contracts,” in *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops*, 2020, pp. 754–761.
- [64] Z. Zhang, Y. Lei, X. Mao, and P. Li, “Cnn-fl: An effective approach for localizing faults using convolutional neural networks,” in *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2019, pp. 445–455.
- [65] X. Li, W. Li, Y. Zhang, and L. Zhang, “Deepfl: Integrating multiple fault diagnosis dimensions for deep fault localization,” in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2019, pp. 169–180.
- [66] Z. Zhang, Y. Lei, X. Mao, M. Yan, L. Xu, and X. Zhang, “A study of effectiveness of deep learning in locating real faults,” *Information and Software Technology*, vol. 131, p. 106486, 2021.
- [67] X. Gu, H. Zhang, D. Zhang, and S. Kim, “Deep api learning,” in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2016, pp. 631–642.
- [68] S. Wang, M. Wen, B. Lin, and X. Mao, “Lightweight global and local contexts guided method name recommendation with prior knowledge,” in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2021.
- [69] A. N. Lam, A. T. Nguyen, H. A. Nguyen, and T. N. Nguyen, “Combining deep learning with information retrieval to localize buggy files for bug reports (n),” in *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2015, pp. 476–481.
- [70] X. Gu, H. Zhang, and S. Kim, “Deep code search,” in *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 2018, pp. 933–944.