Reentrancy Vulnerability Detection and Localization: A Deep Learning Based Two-phase Approach

Zhuo Zhang*
School of Big Data & Software Engineering, Chongqing University
Chongqing, China
zz8477@126.com

Yan Lei†
School of Big Data & Software Engineering, Chongqing University
Chongqing, China
yanlei@cqu.edu.cn

Meng Yan
School of Big Data & Software Engineering, Chongqing University
Chongqing, China
mengy@cqu.edu.cn

Yue Yu
College of Computer, National University of Defense Technology
Changsha, China
yuyue@nudt.edu.cn

Jiachi Chen
School of Software Engineering, Sun Yat-sen University
Guangzhou, China
chenjch86@mail.sysu.edu.cn

Shangwen Wang
College of Computer, National University of Defense Technology
Changsha, China
wangshangwen13@nudt.edu.cn

Xiaoguang Mao
College of Computer, National University of Defense Technology
Changsha, China
xgmao@nudt.edu.cn

ABSTRACT

Smart contracts have been widely and rapidly used to automate financial and business transactions together with blockchains, helping people make agreements while minimizing trusts. With millions of smart contracts deployed on blockchain, various bugs and vulnerabilities in smart contracts have emerged. Following the rapid development of deep learning, many recent studies have used deep learning for vulnerability detection to conduct security checks before deploying smart contracts. These approaches show effective results on detecting whether a smart contract is vulnerable or not whereas their results on locating suspicious statements responsible for the detected vulnerability are still unsatisfactory.

To address this problem, we propose a deep learning based two-phase smart contract debugger for *reentrancy* vulnerability, one of the most severe vulnerabilities, named as ReVulDL: Reentrancy Vulnerability Detection and Localization. ReVulDL integrates the vulnerability detection and localization into a unified debugging pipeline. For the detection phase, given a smart contract, ReVulDL uses a graph-based pre-training model to learn the complex relationships in propagation chains for detecting whether the smart contract contains a *reentrancy* vulnerability. For the localization phase, if a *reentrancy* vulnerability is detected, ReVulDL utilizes interpretable machine learning to locate the suspicious statements in smart contract to provide interpretations of the detected vulnerability. Our large-scale empirical study on 47,398 smart contracts shows that ReVulDL achieves promising results in detecting *reentrancy* vulnerabilities (*e.g.,* outperforming 16 state-of-the-art vulnerability detection approaches) and locating vulnerable statements (*e.g.,* 70.38% of the vulnerable statements are ranked within Top-10).

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**.

KEYWORDS

Smart contract, reentrancy vulnerability, vulnerability detection, fault localization

ACM Reference Format:
Zhuo Zhang, Yan Lei, Meng Yan, Yue Yu, Jiachi Chen, Shangwen Wang, and Xiaoguang Mao. 2022. Reentrancy Vulnerability Detection and Localization: A Deep Learning Based Two-phase Approach. In *37th IEEE/ACM International Conference on Automated Software Engineering (ASE '22), October 10–14, 2022, Rochester, MI, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3551349.3560428

1 INTRODUCTION

Cryptocurrencies and blockchain technologies have experienced a rapid development in industry and academia in recent years. A blockchain is a transaction medium that utilizes a well-designed consensus protocol to ensure transaction security and control the

*Also with School of Information and Communication Technology & Guangzhou College of Commerce, Guangzhou, China.
†Yan Lei is the corresponding author and also with Peng Cheng Laboratory, Shenzhen, China.

ASE '22, October 10–14, 2022, Rochester, MI, USA

ACM ISBN 978-1-4503-9475-8/22/10…$15.00
https://doi.org/10.1145/3551349.3560428

creation of transaction units [13, 64]. A blockchain platform [49] provides networked units with value transfer without trust, and developers deploy smart contracts on the blockchain platform (*e.g.,* the most popular blockchain platform Ethereum [9]) allowing networked peers to make agreements with minimizing trust. Since smart contracts are designed to control digital assets with defined arbitrary rules, they always involve cryptocurrencies worthy of millions of USD. Thus, managing so much wealth under smart contracts will inevitably attract the attention of hackers. Indeed, security problems with smart contracts have led to serious consequences. For instance, the well-known reentrancy smart contract attacks cause millions of dollars loss [1]. As a result, smart contract vulnerabilities are of great necessity to be detected and located before deployment since (1) developers often inject vulnerabilities into smart contracts unconsciously under the condition that they fail to understand the relationships among the cooperating smart contracts; and (2) once smart contracts are deployed to the blockchain, they are difficult to be updated due to their immutability [12].

Many smart contract vulnerability detection approaches have been proposed to tackle the security problems. The conventional vulnerability detection approaches are based on classical static analysis methods and dynamic execution methods [23, 35, 36, 45, 47, 48, 59, 60, 62, 68]. However, these conventional approaches rely heavily on manually crafted heuristics, which are difficult to be summarized for a wide spectrum of vulnerabilities in practice and also suffer from the inherent risk of being error-prone as well as the tricks used by crafty attackers who may surpass these already known rules [13, 64]. With the remarkable achievements of deep learning in many fields, researchers recently explore the use of deep neural networks to detect smart contract vulnerabilities [26, 43, 51, 58, 72] to address the aforementioned limitations. These approaches either treat the source code as a text sequence or cast the rich control and data flow semantics of the source code into graphs for the learning process, and have achieved promising performances on vulnerability detection for smart contracts. Although many deep-learning-based approaches are generally effective in determining whether an entire given contract is vulnerable or not, their results on providing the explanation of a context with which the deep learning model have made the decision (a.k.a, the *vulnerability localization* in this paper) are still unsatisfactory. This is a critical capability of deep-learning-based debugging tools since the number of code statements of smart contracts usually range from dozens to thousands. Thus, it is still time-consuming and error-prone for developers to manually locate the faulty statements even if they are aware of the existence of the vulnerability. In addition, vulnerability detection and localization are two interactive tasks. Specifically, vulnerability localization usually relies on the vulnerability symptoms (*e.g.,* execution traces) discovered by the detection phase to analyze and pinpoint the root cause of vulnerability, and is thus a natural extension of detection.

Thus, we propose a deep learning based two-phase debugging approach for smart contracts that explicitly targets the *reentrancy* vulnerability, ReVulDL: Reentrancy Vulnerability Detection and Localization. *Reentrancy* vulnerability is one of the most severe vulnerabilities in smart contracts [55, 67], occurring when the attacker drains funds from the target by recursively calling the target to withdraw function. It has also caused huge financial losses in recent years, *e.g.,* the famous DAO incident causing 3.6 Million Ether stolen and 60 million USD loss [56] due to the *reentrancy* attack. Therefore, debugging this type of vulnerability has attracted the attentions from many researchers [65, 67]. Our ReVulDL adopts deep learning techniques (*i.e.,* a graph-based pre-training model and interpretable machine learning) to integrate *reentrancy* vulnerability detection and localization into a unified debugging pipeline, for pinpointing the root cause of a *reentrancy* vulnerability.

For *reentrancy* vulnerability detection phase, ReVulDL presents a graph-based pre-training model to learn the complex relationships in propagation chains to detect whether a smart contract is vulnerable to *reentrancy*. Specifically, given a smart contract *SC*, ReVulDL first constructs propagation chains CH_{SC} by leveraging the data flow graph of the source code of *SC*; then customizes a graph-based pre-training model to learn the complex relationships (*i.e.,* data dependencies) in the propagation chains CH_{SC} that may lead to *reentrancy* vulnerability; and finally uses the trained model to determine whether the smart contract *SC* contains a *reentrancy* vulnerability or not.

For *reentrancy* vulnerability localization phase, if a *reentrancy* vulnerability is detected in *SC*, ReVulDL utilizes the interpretable machine learning (*i.e.,* post-hoc interpretability method) on the detected vulnerability information to locate suspicious statements that might be responsible for the detected smart contract vulnerability. Specifically, ReVulDL first prepares the detected vulnerability information (*i.e.,* the trained model with its decision (vulnerable or not) and the smart contract *SC* in the test dataset with its propagation chains CH_{SC}) for localization; then leverages post-hoc interpretability method [6, 21, 69] to seek the interpretation subchains $minCH_{SC}$ which is the minimal propagation chains in CH_{SC} that eventually leads to the *reentrancy* vulnerability, where the basic idea is that if an edge e_{SC} in CH_{SC} is crucial for the trained model to identify *SC* to be vulnerable, e_{SC} should be included in $minCH_{SC}$; and finally uses the crucial variables and their propagation chains in $minCH_{SC}$ to locate the suspicious statements potentially responsible for the detected *reentrancy* vulnerability.

We design and conduct large-scale experiments on the widely-used dataset (*i.e., SmartBugs Wild Dataset*[25]) with 47,398 real-world smart contracts. The experimental results show that our approach is effective in both detecting *reentrancy* vulnerability (*e.g.,* the *F1* score of 0.93 outperforms those of 16 state-of-the-art baselines) and locating the vulnerable statements (*e.g.,* 20.38%, 44.05%, 58.99%, and 70.38% of the buggy statements are ranked within *Top-1*, *Top-3*, *Top-5*, and *Top-10* respectively). Such performances are promising considering that a number of vulnerabilities in our test set are comparatively complex, involving multiple buggy statements across functions.

The main contributions of this paper can be summarized as:

• We present (1) a *reentrancy* vulnerability detection approach which models propagation chains as graphs and learns the complex relationship from the graph level; and (2) a statement-level *reentrancy* vulnerability localization approach which customizes interpretable machine learning to identify suspicious statements.

• Based on the above two components, we implement a deep learning based two-phase smart contract debugger, ReVulDL, that

[1] https://moralis.io/what-is-reentrancy-reentrancy-smart-contract-example/

not only determines if a smart contract contains a *reentrancy* vulnerability but also pinpoints the root cause of the vulnerability if any. [2]

• We perform large-scale empirical evaluations and the experiment results demonstrate the effectiveness of ReVulDL on both vulnerability detection and localization tasks.

The structure of the rest paper is organized as follows. Section 2 introduces background. Section 3 depicts ReVulDL. Section 4 presents our empirical study. Section 7 summarizes related work. Section 8 concludes the whole study and mentions future work.

2 BACKGROUND

2.1 Interpretable Machine Learning

Interpretable machine learning tackles the important problem of explaining how a complex machine learning model achieves a particular decision and makes up for the deficiency of transparency behind behaviors of learning models [22]. It enables machine learning models to explain or present their behaviors to humans in an understandable way [19]. Generally, there are two interpretable machine learning techniques: intrinsic interpretability and post-hoc interpretability [46]. Intrinsic interpretability refers to machine learning models that are considered interpretable due to their simple structure [11], *e.g.,* decision tree, rule-based model, linear model. The constructed intrinsic interpretation models either are globally interpretable models which offer a certain extent of transparency inside a model or locally interpretable models that provide rationals for a specific prediction. In contrast, post-hoc interpretability refers to a model-agnostic model which provides the explanation after another model is trained [5]. Post-hoc methods are valuable in legal proceedings for their inherent ability in explaining the model after training [20]. They aim to provide an understanding about what knowledge has been acquired and which part of the learned representations or features in the interpreted model is responsible for the predicted result, without changing the underlying model.

In our study, we utilize the interpretable machine learning for localizing the faulty statements in smart contracts. Specifically, the vulnerability localization can be regarded as the process of seeking the explanation of why the decision is made by the trained model of vulnerability detection. Thus, ReVulDL uses the post-hoc interpretability to explain what statements are suspicious for causing the detected vulnerability.

2.2 Propagation Chain

A propagation chain refers to the existence of a code sequence among a certain number of specified program code snippets [28]. In this sequence, there are direct or indirect data/control dependencies between any two adjacent code snippets. Given code snippet a and code snippet b, there may be one or more propagation chains between a and b, which is called the propagation chain set of specified code snippets a and b (denoted as PC(a, b)). For each program snippet, there are several propagation chains affected by it and a certain number of propagation chains affecting it. In software debugging, a defect propagation chain refers to a code sequence from the defect code to the program failure output. One or more code snippets in this sequence have errors in the program state (such as variable value) during execution. There are one or more defect propagation chains from program defect code d to program failure code f, and the set of the defect propagation chains is called defect propagation chain set (EPC(d, f)). EPC(d, f) is a subset of the propagation chain set PC (d, f) from code snippet d to program failure code f. For a smart contract, there are one or more propagation chains related to vulnerability code snippets of the smart contract, which could be learned by deep learning models. Propagation chains can be constructed by data flow relationships or control flow relationships, providing crucial code semantic information for program analysis and comprehension [2, 29, 30].

In this paper, we use data flow relationships to construct propagation chains since they are the same under different abstract grammars for the same source code and easier for deep-learning models to learn (see the evaluation on other types of relationships in Section 4.6). In data flow graph, nodes represent variables in a smart contract while edges denote the dependency relations among these variables. For example, after constructing the data flow graph of $c = a + b$, it would be clear that the value of c depends on a and b, providing a new perspective for understanding the semantics of the variable c. Therefore, ReVulDL takes the advantage of the relationships in the propagation chains of a smart contract, and uses a graph-based pre-training model to learn the relationships for detecting *reentrancy* vulnerability.

Victim contract

```
1 contract Private_accumulation_fund
2 {
3     mapping (address => uint) public balances;
4     uint public MinDeposit = 1 ether;
5     Log TransferLog;
6     function Private_accumulation_fund(address _log)
7     public
8     {
9         TransferLog = Log(_log);
10    }
11    function Deposit()
12    public payable
13    {
14        if(msg.value > MinDeposit)
15        {
16            balances[msg.sender]+=msg.value;
17            TransferLog.AddMessage(
18                    msg.sender,msg.value,"Deposit");
19        }
20    }
21    function CashOut(uint _am)
22    public payable
23    {
24        if(_am<=balances[msg.sender])
25        {
26            if(msg.sender.call.value(_am)())
27            {
28                balances[msg.sender]-=_am;
29                TransferLog.AddMessage(
30                        msg.sender,_am,"CashOut");
31            }
32        }
33    }
34    function() public payable{}
35 }
```

Attack contract

```
1 contract Attack
2{
3    Private_accumulation_fund public entrance;
4    constructor(address _target) public
5    {
6        entrance = Private_accumulation_fund(_target);
7    }
8    function attack() payable
9    {
10       entrance.CashOut(0.5 ether);
11   }
12   function() public payable
13   {
14       // re-enter Private_accumulation_fund
15       entrance.Cashout(0.5 ether);
16   }
17 }
```

Figure 1: A Reentrancy Vulnerability Example.

2.3 Motivating Example

Figure 1 shows a *reentrancy* vulnerability example. The contract *Private_accumulation_fund* is a real contract deployed on Ethereum in SmartBugs Wild Dataset[25]. *Reentrancy* vulnerability is one of the most dangerous smart contract vulnerabilities that has led to huge amounts of financial losses [55, 56, 67]. Attackers could use the mechanism of call and callee statements to stole balance (ether). When the victim contract use call-statement to call attack contract, the attack contract use callee-statement to call the caller and enter the caller in victim contract again, and consequently the balance is stolen. The attackers use this mechanism to eventually withdraw a large amount of Ethers from victim contract's balance. As shown in Figure 1, there are two smart contract: the victim contract *Private_accumulation_fund* and the attack contract *Attack*. The *Attack* contract calls the function *attack* at line 8, and then calls the *Cashout* function at line 21 of the contract *Private_accumulation_fund*. At

[2]Our replication package including the source code, datasets and running examples is publicly available at https://github.com/toolstemp/IAcontract.

the same time, *Private_accumulation_fund* will execute function *Cashout* at line 21 and send *ether* to contract *Attack* with the call-statement at line 26. When the call-statement at the line 26 is executed, the fallback function[3] of the contract *Attack* will be executed. Thus, the fallback function at line 12 of the contract *Attack* responds to the transfer of the contract *Private_accumulation_fund*, and the contract *Attack* will keep withdrawing the ether from the contract *Private_accumulation_fund* until the gas runs out. In the meantime, the line 28 of the contract *Private_accumulation_fund* deducts the balance of contract *Attack* once while the record in block chain is only the first withdrawal.

The reason for the *reentrancy* vulnerability lies in the fact that the atomic transfer operation becomes non-atomic by transferring money first and then deducting the balance. It gives the attacker the opportunity to reentry and get multiple transfers, but only one balance deduction is recorded on the block chain. Thus, the vulnerable statement is located at line 28 for the fact that there is a balance operation at line 28 after the non-atomic transfer operation *call.value* at line 26. Other code snippets such as statements in the function *Private accumulation fund* and *Deposit* have neither non-atomic transfer operation nor data propagation with the function *Cashout*. Consequently, what the programmers really expect to seek out are the statements at line 26-28 and other ones are of limited usefulness to locate the *reentrancy* vulnerability.

Therefore, regarding to the number of smart contract statements ranging from dozens to thousands, it is necessary to identify what statements are responsible for a detected vulnerability. With the help of interpretable machine learning, ReVulDL integrates two phases: vulnerability detection and localization to pinpoint what specific statements are potentially responsible for the detected vulnerability. Specifically, based on the vulnerability symptoms (*e.g.,* the trained model with its decision (vulnerable or not) and the smart contract in the test dataset with its full propagation chains) discovered by vulnerability detection, ReVulDL uses interpretable machine learning to find a convergent subset of propagation chains with corresponding statements related to the detected vulnerability, and provides explanation on why the trained model makes the decision by outputting a ranking list of suspicious statements potentially responsible for the detected vulnerability.

3 APPROACH

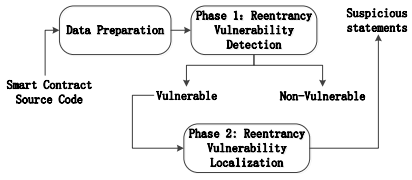

Figure 2: Workflow of ReVulDL.

This section depicts our deep learning based two-phase approach ReVulDL: Reentrancy Vulnerability Detection and Localization, to pinpoint what specific statements that might be involved in the detected *reentrancy* vulnerability. ReVulDL uses a graph-based pre-training model to learn the relationships among the propagation chains to detect *reentrancy* vulnerability, and utilizes interpretable machine learning (*i.e.,* post-hoc interpretability) to provide an interpretation of the decision of the detection model to locate suspicious statements relevant to the detected vulnerability.

Figure 2 shows the workflow of ReVulDL, consisting of three parts: (1) *Data Preparation*, which constructs the representation for smart contract code features and extracts propagation chains for the next two phases (*i.e.,* vulnerability detection and localization). (2) *Reentrancy Vulnerability Detection* phase, which detects whether a smart contract contains a *reentrancy* vulnerability based on the pre-trained model. (3) *Reentrancy Vulnerability Localization* phase, which locates the suspicious statements responsible for the detected *reentrancy* vulnerability based on post-hoc interpretability.

3.1 Data Preparation

ReVulDL first needs to prepare data (*i.e.,* the representation of smart contract code features and propagation chains) for the two phases (*i.e.,* vulnerability detection and localization). For a contract, ReVulDL extracts different types of data via the following two steps:

Representation of smart contract source code. ReVulDL captures the content of a smart contract source code in terms of the sequence of tokens. The tokens of each statement are collected and broken down into sequence. ReVulDL removes the sub-tokens with one character to avoid the influence of noises. Specifically, suppose there is a contract A. From A, the source code set is $C = \{c_1, c_2, ..., c_{m_c}\}$ and variable set is $V = \{v_1, v_2, ..., v_{m_v}\}$. Then, the two sets are concatenated into a sequence $I = \{[CLS], C, [SEP], V\}$, where [CLS] is a special token in front of the two sets and *[SEP]* is a special notation to split the source code set C and the variable set V. For the representation of A, the sequence I is transformed into an input vector, in which each token is represented by the embeddings of token itself and its position.

Construction of propagation chains. ReVulDL first parses the smart contract source code into an abstract syntax tree (AST); then extracts data flow relationships from the AST; finally constructs propagation chains from the dataflow relationships. Specifically, ReVulDL converts the source code set is $C = \{c_1, c_2, ..., c_{m_c}\}$ to AST using treesitter [44]. Since solidity is the most common language used in smart contracts, ReVulDL follows the JoranHonig's grammar [32], and customizes treesitter to solidity language [15] for facilitating data flow relationship construction. The AST from the source code C contains syntax information of the contract source code while the terminals (leaves) are used to gain the variable set $V = \{v_1, v_2, ..., v_{m_v}\}$. For each variable in the variable set $V = \{v_1, v_2, ..., v_{m_v}\}$, marked as v_i, we create a direct edge $\epsilon = \langle v_i, v_j \rangle$ from v_i to v_j, indicating that the value of j-th variable comes from or is computed from i-th variable. After creating edges for every variable, we could filter out the variables without edge flowing to them as the initial nodes and pull out chain structures from these variables. These chain structures are propagation chains for the smart contract source code.

3.2 Reentrancy Vulnerability Detection

For *reentrancy* vulnerability detection phase, ReVulDL customizes a graph-based pre-training model to learn the contract code features and propagation chains acquired from *Data Preparation*. Pre-trained models have acquired big success in tasks of natural language processing (NLP) [17, 34, 42, 52, 53], inspiring the use on

[3] A fallback function is executed when a contract receives Ether without any additional data.

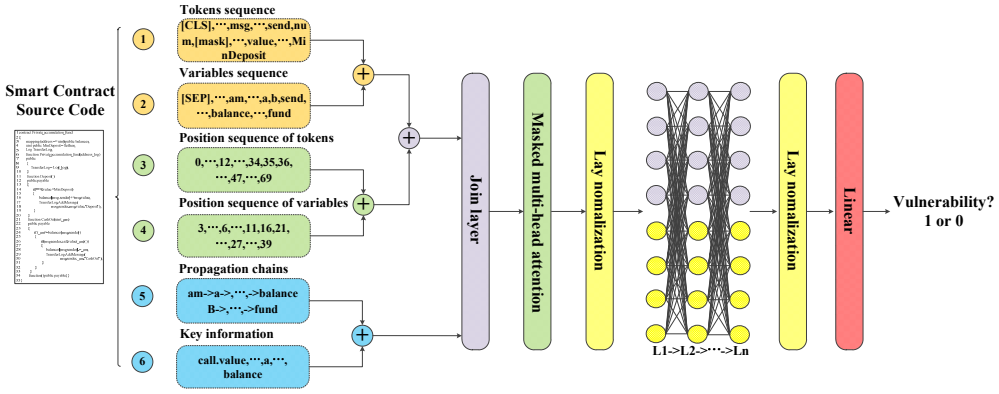

Figure 3: Vulnerability Detection Architecture of ReVulDL.

programming languages to promote the development of code intelligence [8, 24, 37, 38, 57]. There are widely used models such as GPT [52] and BERT [17] for NLP and Scelmo [38], CodeBert [24], C-Bert [8] and GPT-C [57] for programming languages. These pre-trained techniques on programming languages pre-train on a large unsupervised code corpus, and fine-tune on downstream tasks, *e.g.*, code search [29], code completion [39], and code summarization [3].

Compared with other techniques which consider a source code as a sequence of tokens, GraphCodeBERT [29] regards the inherent structure of code. It is trained on the CodeSearchNet dataset[33], which includes 2.3 millions functions of six programming languages paired with natural language documents. In pre-training stage, GraphCodeBERT uses a semantic-level structure that is less complex and does not bring an unnecessarily deep hierarchy, the property of which helps the model show better capability and performance. During pre-training, the first task is masked language modeling [17], which aims to learn representation from the source code; the second task is edge prediction for learning representation from code graph; The third is variable-alignment, which aims to align the representation between source code and graph and to predict where a variable is identified from. Thus, we use GraphCodeBERT as backbone to learn the propagation chains of smart contract code and perform the vulnerability detection task. In view of the success of pre-trained models in the field of natural language processing, we seek a new perspective probably benefiting *reentrancy* vulnerability detection phase in smart contract. Therefore, ReVulDL customizes the vulnerability detection model from a graph-based pre-trained model GraphCodeBERT [29], which processes programming language based on Transformer neural architecture [63].

Figure 3 shows the vulnerability detection architecture of ReVulDL, consisting of 6 parts, which are input units, join layer, masked multi-head attention layer, layer normalization layers, n transformer layers and linear layers. There are 6 input units in ReVulDL constructed from *Data Preparation*: token sequence, variable sequence, position sequence of tokens, position sequence of variables, propagation chains and key information. The first 4 units are the representations of smart contract source code. As previously discussed, we concatenate token sequence set and variable sequence set into a sequence $I = \{[CLS], C, [SEP], V\}$ while concatenating position sequence of tokens and position sequence of variables into another sequence. $[CLS]$ is a special token in front of the two sets and $[SEP]$ is a special notation to split the source code C and the variable set V. We transform these two sequence into an input vector X^0 as the representation of the smart contract source code. X^0 contains two embeddings, where one denotes token sequence set and variable sequence while the other represents position of tokens and position of variables. We obtain propagation chains from AST with data flow relationships and denote them as $PC = \{pc_1, pc_2, ..., pc_{m_{pc}}\}$. For pc_i $(i \in \{1, 2, ..., m_{pc}\})$, $pc_i = (V, E)$, where $V = \{v_1, v_2, ..., v_k\}$ is a variable set, and $E = \{\epsilon_1, \epsilon_2, ..., \epsilon_{k'}\}$ is a directed edge set indicating where the value of each variable comes from or computed from. Key information is extracted from the source code as $V' = \{v'_1, v'_2, ..., v'_l\}$, which are at the same line with *call.value* or have a direct data flow relation to the line's variables. Then, we expand variables on the propagation chains' path through $v'_i (i \in \{1, 2, ..., l\}$ to V' and remove the discrete elements in V' through which none of the edges in propagation chains pass. Finally, V' is rebuilt and we could remove redundancies of propagation chains which are irrelevant to elements in V'. The cropped propagation chains are critical for *reentrancy* vulnerability.

The 6 input units are fed to the join layer for generating the contextual representation r^0. After the join layer, there are n structurally equivalent transformer layers which are L_1 to L_n for r^0 to go through, $r^i = transformer_i(r^{i-1}), i \in [1, n]$. The calculation process of $transformer_i$ corresponds to Equation (1) and Equation (2). In Equation (1), the vector r^{i-1} will first generate the vector D^i after a multi-headed self-attentive operation [63]. Then in Equation (2), D^i will output the vector r^i after a feed-forward layer.

$$D^n = LayerNorm(MultiAttn(r^{n-1}) + r^{n-1}) \tag{1}$$

$$r^n = LayerNorm(FeedForwardNet(D^n) + D^n) \tag{2}$$

In Equation (1) and Equation (2), *MultiAttn* is a multi-headed self-attention mechanism, *FeedForwardNet* is a two layers feed forward network, and *LayerNorm* represents a layer normalization operation. For the n-th transformer layer, the output $\hat{r}^n$ of a multi-headed self-attention (*i.e., MultiAttn* in Equation (1)) is computed by Equation (3), Equation (4) and Equation (5).

$$Q_i = r^{n-1}{W_i}^Q, K_i = r^{n-1}{W_i}^K, V_i = r^{n-1}{W_i}^V \tag{3}$$

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

$$\hat{r}^n = [head_1; ...; head_m]{W_n}^O \tag{5}$$

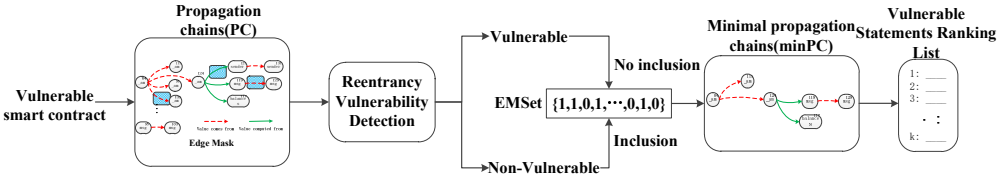

Figure 4: Vulnerability Localization Architecture of ReVulDL.

Where, the previous layer's output $r^{n-1} \in \mathbb{R}^{|I| \times d_h}$ 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 $W_n^Q \in \mathbb{R}^{d_h \times d_h}$ 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 -∞.

According to GraphCodeBERT [29], we use graph-guided masked attention function to model variable dependency relations and incorporate graph structure into Transformer. Equation (6) shows the mask matrix M to demonstrate graph-guided masked attention.

$$M_{ij} = \begin{cases} 0 & \text{if } q_i \in \{[CLS], [SEP]\} \\ & \text{or } q_i, k_j \in C \\ & \text{or } \langle q_i, k_j \rangle \in E \cup E' \\ -\infty & \text{otherwise} \end{cases} \tag{6}$$

The key k_i attended by the query q_j could be avoided by adding an infinite negative value to the attention score ${q_j}^T k_i$ so that the attention weight becomes zero after using the *softmax* function (*i.e.,* Equation (4)). In Equation (6), C is the source code set as declaration before. For each propagation chain $pc_i (i \in \{1, 2, ..., m_{pc}\})$, E is a directed edge set indicating where the value of each variable comes from or computed from. The query of node v_i is allowed to pay attention to a node-key of node v_j if there is a direct edge between node v_i and node v_j (*i.e.,* $\langle v_i, v_j \rangle \in E$) or they are the same node (*i.e.,* $i = j$). Otherwise, the attention is masked, adding the attention score to an infinitely large negative value. E' is a set to denote the relation between smart contract source code tokens and variables of the propagation chain, where $\langle v_i, c_j \rangle \langle c_j, v_i \rangle \in E'$ if the variable v_i is identified from the code token c_j. We allow query of node v_i and node-key of c_j attend each other if and only if $\langle v_i, c_j \rangle \langle c_j, v_i \rangle \in E'$. At the end of the model, we add a linear classifier and use the *Sigmoid* function to output the predicted probabilities $\hat{y}$ in Equation (7).

$$\hat{y} = Sigmoid(\hat{r}^n) \tag{7}$$

In summary, the vulnerability detection phase seeks to find whether a given smart contract contains *reentrancy* vulnerability by fine-tuning the detection model, which is fed with a large number of smart contract source code and corresponding propagation chains, together with their ground truth labels.

3.3 Reentrancy Vulnerability Localization

For *reentrancy* vulnerability localization phase, ReVulDL takes input as the vulnerability symptoms (*i.e.,* the trained vulnerability detection model, propagation chains of smart contract source code, detection result and prediction score) acquired from *Reentrancy Vulnerability Detection* phase; then adopts interpretable machine learning (post-hoc interpretability) to identify interpretable sub-propagation chains providing explanation why the detection model makes the vulnerable decision; finally outputs a ranking list of suspicious statements responsible for the detected vulnerability.

Figure 4 shows the vulnerability localization architecture of ReVulDL, the aim of which is to find out minimal propagation chains *minPC* that minimize the difference in the prediction scores between using the original propagation chains *PC* and using the minimal propagation chains *minPC*. With the help of GNNExplainer[70] which treats the task of finding subgraphs as an edge-mask learning task, we derive interpretation sub-propagation chains by learning an edge-mask set *EMSet* and masking-out the edges in the original propagation chains. Since an edge is cut in the original propagation chains *PC*, vulnerable statements localization model of ReVulDL checks whether the vulnerability detection model outputs the same prediction result or not. If yes, the edge is not crucial and need not to be included in *minPC*. Otherwise, the edge is important and should be included in *minPC*. In view of the untractable characteristics of edge-mask, we use a learning approach for *EMSet*. The task could be formulated as maximizing the mutual information (*MI*) between the minimal propagation chains *minPC* and the original propagation chains *PC* (see Equation (8)).

$$\max_{\text{minPC}} MI(\hat{y}, minPC) = H(\hat{y}) - H(\hat{y}|G = minPC) \tag{8}$$

Where, $\hat{y}$ is the output score of the vulnerability detection model and $H(\hat{y})$ is constant for the trained vulnerability detection model (denoted as TVDM) accordingly. Thus, maximizing the *MI* value for *minPC* is equivalent to minimizing conditional entropy $H(\hat{y}|G = minPC)$.

$$-\mathbb{E}_{\hat{y}|minPC}[\log P_{TVDM}(\hat{y}|G = minPC)] \tag{9}$$

Equation (9) measures how much uncertainty remains about the outcome $\hat{y}$ under the condition that $G = minPC$. Direct optimization of Equation (9) is not tractable, we further treat *minPC* as a random graph variable *minPC'*. The objective of Equation (9) changes to the following Equation (10) and (11).)

$$\min_{minPC'} \mathbb{E}_{\min PC \sim minPC'} H(\hat{y}|G = minPC) \tag{10}$$

$$\min_{minPC'} H(\hat{y}|G = \mathbb{E}_{minPC'}[minPC]) \tag{11}$$

According to Jensen's inequality[10], we transform Equation (10) to Equation (11). The conditional entropy in Equation (10) can be optimized by replacing $\mathbb{E}_{minPC'}[minPC]$ and masking with *EMSet* on the input graph *minPC*. Finally, learning the mask set *EMSet* is available according to [70] and the resulting sub-propagation chains *minPC* could directly be utilized as an interpretation for vulnerability statements localization. With the generated sub-propagation chains *minPC*, ReVulDL can calculate each element's suspiciousness in *minPC*. ReVulDL initially assigns the elements (*i.e.,* variables) in *minPC* with the same suspiciousness value. There are two types of edges in *minPC*. One is *values comes from* that denotes a variable's value comes from another one; the other one is *values computed from* which means that a variable's value is computed from another one. Then, based on the types of edges in *minPC*, ReVulDL calculates the suspiciousness of each element in *minPC* in the two cases. In the case of *values comes from* edge type, ReVulDL assigns the maximum suspiciousness value of the data-dependent (*values comes from*) elements; in the case of *values computed from* edge type,

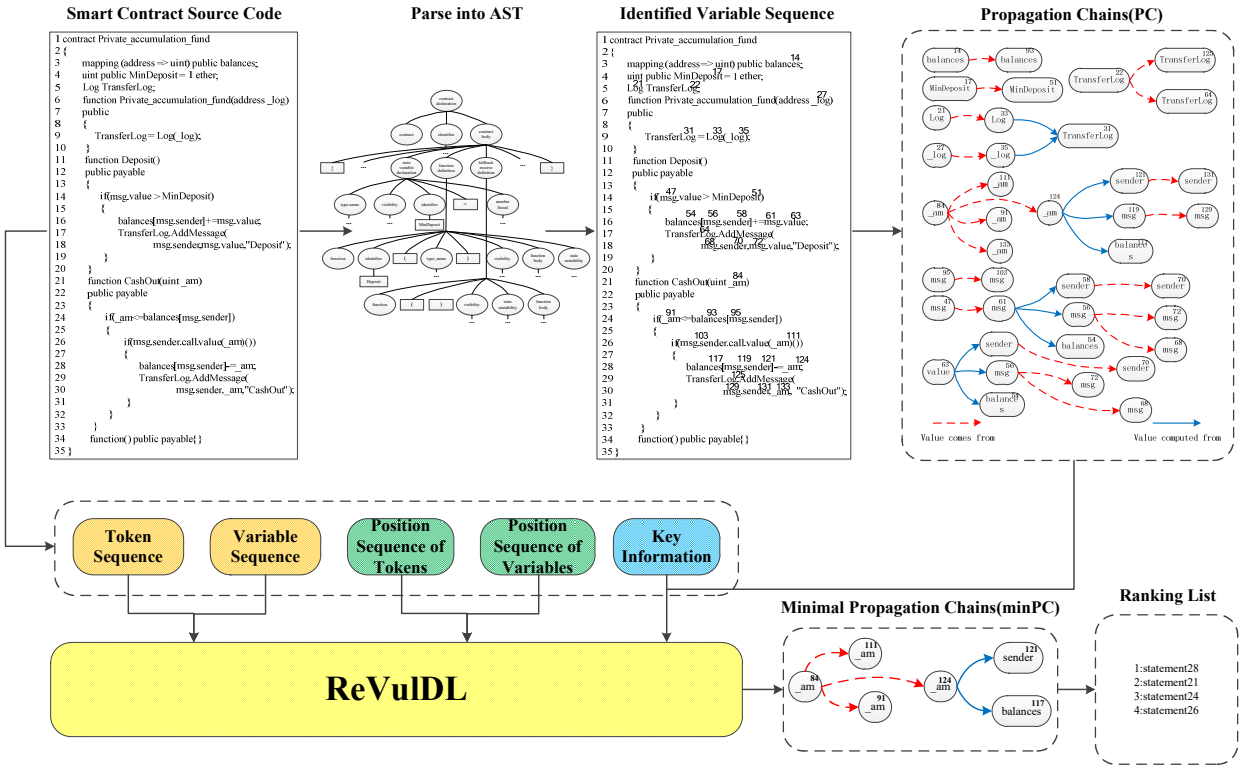

Figure 5: An example illustrating ReVulDL.

ReVulDL adds up the suspiciousness values of the data-dependent (*values computed from*) elements and assign the sum to the elements as the new suspiciousness value. For example, the data dependence (*values computed from*) of e_i contains e_j and e_k, meaning that there are two direct edges of *values computed from* type to e_i in *minPC*: e_j to e_i and e_k to e_i. Suppose that the current suspiciousness values of e_i, e_j and e_k are s_i, s_j and s_k respectively. We add the suspiciousness values s_i, s_j and s_k to *sum*, *i.e.,* $sum = s_i + s_j + s_k$. Then we assign the value of *sum* to s_i, which is the new suspiciousness value of e_i .

According to the above method, we rank the elements in *minPC* in descending order of the suspiciousness values and obtain the final ranking list of elements. Since each element in *minPC* denotes a variable in smart contract source code, we could replace the variable in the ranking list with the statement that the variable belongs to. Finally, ReVulDL could obtain a ranking list of vulnerable statements in descending order of suspiciousness. As a reminder, for a vulnerable contract is incorrectly identified as non-vulnerable in the first phase, we set the rank of this contract to be null.

3.4 Illustrative Example

To show how ReVulDL works, Figure 5 illustrates a real smart contract *SC* deployed on Etherium with a *reentrancy* vulnerability. The top left of Figure 5 shows the source code of *SC*. As mentioned in Section 2.3, the reason why *reentrancy* vulnerability exists is that the atomic transfer operation becomes non-atomic by transferring money first (at $statement_{26}$) and then deducting the balance (at $statement_{28}$) so that the hacker may make multiple transfers but only one deduction of the balance is recorded. Thus, we recognize $statement_{28}$ as the vulnerable statement.

ReVulDL first extract token sequence, variable sequence, position sequence of tokens, position sequence of variables and key information from the source code of *SC*. Then, ReVulDL acquires the propagation chains by parsing *SC* into AST by modified treesitter and identifying variable sequence from AST. For example, the variable *_am* at the line 21 is ranked 84th in the sequence of variables while at the line 28, the variable *_am* is the 124th in the variable sequence. They have the same name *_am* in a propagation chain but are different variables. We take each variable as a node of the corresponding propagation chain, and a direct edge indicates the value of a variable comes from or is computed from another one. Taking *TranferLog = expr* at the line 9 as an example, the edges from all variables in *expr* to *TranferLog* are added into the propagation chain and labeled as "*computed from*". Meanwhile, the edges like from 27th *_log* to 35th *_log* are added into the propagation chain and labeled as "*comes from*". We utilize the dependency relation between variables of data flow to construct propagation chains. In Figure 5, there are several propagation chains *PC*, in which the red dotted line denotes "*values comes from*" while blue solid line represents "*values computed from*". The 6 units token sequence, variable sequence, position sequence of tokens, position sequence of variables, key information and propagation chains are inputted into ReVulDL. After *reentrancy* vulnerability detection and localization, we acquire the minimal propagation chains *minPC*. In Figure 5, there are 6 variables in *minPC*, from which we could compute the suspiciousness of elements according to method in Section 3.3 and output a ranking

list $\{\_am^{124}, \_am^{84}, \_am^{91}, sender^{111}, msg^{117}, balances^{121}\}$. Finally, since these variables belong to their specific statements, we could obtain the ranking list of the statements in descending order of suspiciousness $\{statement_{28}, statement_{21}, statement_{24}, statement_{26}\}$. The vulnerable $statement_{28}$ is ranked 1st.

4 EMPIRICAL EVALUATION

4.1 Research questions

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

RQ1: How effective is ReVulDL in detecting* reentrancy *vulnerability in smart contracts? It is worthwhile to investigate whether ReVulDL outperforms the state-of-the-art vulnerability detection methods in smart contracts. We compare ReVulDL with the 16 state-of-the-art *reentrancy* vulnerability detection approaches.

RQ2: How effective is ReVulDL in locating vulnerable statements in smart contracts? We study how well ReVulDL performs on smart contract vulnerable statements localization. We compare ReVulDL with the seven state-of-the-art baselines.

RQ3: How the different modules contribute to the ReVulDL? We investigate how the different modules (*i.e.,* propagation chains and graph-based pre-trained model) contribute to the ReVulDL. We design ablation experiments to answer this research question.

4.2 Experimental Setup

Benchmarks We empirically evaluate ReVulDL on all the Ethereum smart contracts that have source code verified by Etherscan[25]. Thus, our experiments use the large-scale popular dataset *SmartBugs Wild Dataset* [25], which contains 47,398 real smart contracts.

Experimental settings For RQ1, we compare ReVulDL with the 16 state-of-the-art vulnerability detection approaches, where the eight approaches are conventional ones withoutng deep learning (*i.e.,* Honeybadger [61], Manticore [47], Mythril [48], Osiris [60], Oyente [45], Securify [62], Slither [23] and SmartCheck [59]), and the other eight are deep-learning-based ones (*i.e.,* GCN [43], VanillaRNN [43], Peculiar [65], LSTM [43], GRU [43], DR-GCN [72], TMP [72] and CGE [43]).

For RQ2, although the eight state-of-the-art deep learning based approaches focus on vulnerability detection and cannot localize vulnerable statements, we still seek to evaluate ReVulDL with more comparisons. Regrading that ReVulDL is a deep learning based two-phase approach, we can use eight deep learning based detection approaches instead of our detection approach in the detection phase to perform vulnerability localization and compare ReVulDL with them to evaluate localization effectiveness.

For RQ3, to investigate the contribution of graph-based pre-trained model, we normalize the parameters of the pre-trained model and retrain it with the original input. As for the propagation chains using data flow relationships, we use the other two types of relationships (*i.e.,* control flow relationships, and the combination of control and data flow relationships), instead of data flow relationships used by ReVulDL, to construct the propagation chains to perform the comparison.

Environment The physical environment of the experiments is on a computer containing a CPU of Intel I7-9700 with 64G physical memory, and two 12G GPUs of NVIDIA TITAN X Pascal. The operating system is Ubuntu 18.04. We conducted the statistical comparison on MATLAB R2016b.

Parameter settings ReVulDL uses the adam optimizer, and applies a grid search for the best settings of hyper-parameters: the learning rate is 2e-5; the batch size of the training dataset is 2; the batch size of the validation dataset is 32; the gradient accumulation step is 1; the adam epsilon is 1e-8. For splitting the original dataset, we follow the conventional strategy by randomly selecting 20% of them as the training set, 10% of them as the validation set and the other 70% as the testing set. The ground truth labels for contracts are provided by the prior work of the experts [43] and the labeled dataset has been open sourced in our online repository.

4.3 Evaluation Metrics

We adopt seven widely used metrics, where *Precision*, *Recall*, and *F1-score* [59, 72] are the metrics for vulnerability detection while *Top-N Accuracy* [40, 50], *Mean Average Rank (MAR)* [41], *Mean First Rank (MFR)* [41], and *Relative Improvement (RImp)* [16, 71] are the metrics for vulnerability localization.

Precision, Recall, and F1 *Precision = True Positive/(True Positive + False Positive)*, *Recall = True Positive/(True Positive + False Negative)*, $F1 = 2 \times Precision \times Recall/(Precision + Recall)$. To evaluate the overall performance of our approach equally across the class-imbalanced dataset, we adopt the *macro* way [65] to compute the three metrics for the contracts with and without vulnerabilities, respectively, and then use the average value as the final result.

Top-N Accuracy It denotes the percentage of vulnerabilities located within the first N position of a ranked list of all statements in descending order of suspiciousness returned by a vulnerability localization approach.

Mean Average Rank (MAR) It is the mean of the average rank of all vulnerabilities using a vulnerability localization approach.

Mean First Rank (MFR) For a vulnerability with multiple vulnerable statements, locating the first one is critical since the others may be located after that. *MFR* is the mean of the first vulnerable statement's rank of all vulnerabilities using a vulnerability localization approach.

Relative Improvement (RImp) It is to compare the total number of statements that need to be examined to find all vulnerable statements using ReVulDL versus the number that need to be examined by using other vulnerability localization approaches. A lower value of *RImp* shows better improvement.

4.4 Vulnerability Detection Results (RQ1)

Precision, Recall, F1 and number of detected vulnerabilities We compare ReVulDL with 16 state-of-the-art smart contract vulnerability detection approaches. Table 1 shows the detection performance (*i.e., Recall*, *Precision* and *F1*) of different approaches, and Figure 6 further visualizes the quantitative results of the performance distribution. In addition, Table 1 shows the number of detected vulnerabilities of each approach, *i.e.,* the column 'Number'.

As shown in Table 1 and Figure 6, we could observe that ReVulDL obtains the best results of *Precision*, *Recall*, *F1*, and the number of detected vulnerabilities among the 16 baseline approaches, *i.e.,* ReVulDL achieves 0.92 *Precision*, 0.93 *Recall*, and 0.93 *F1*, 790 *detected vulnerabilities*, significantly improving the 16 baselines.

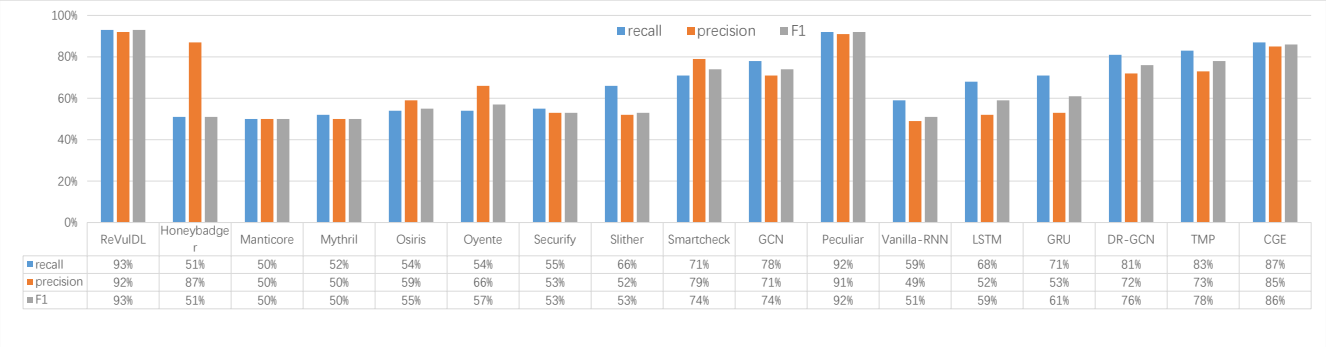

Figure 6: *Recall*, *Precision* and *F1-score* distribution of ReVulDL over 16 baselines.

Table 1: Detection performance comparison of ReVulDL and the 16 baselines in terms of *Recall*, *Precision*, *F1* and the number of detected vulnerabilities.

Approach	Reentrancy			
	Recall	Precision	F1	Number
Honeybadger	0.51	0.87	0.51	19
Manticore	0.50	0.50	0.50	17
Mythril	0.52	0.50	0.50	50
Osiris	0.54	0.59	0.55	65
Oyente	0.54	0.66	0.57	69
Securify	0.55	0.53	0.53	341
Slither	0.66	0.52	0.53	293
Smartcheck	0.71	0.79	0.74	88
GCN	0.78	0.71	0.74	595
Vanilla-RNN	0.59	0.49	0.51	432
Peculiar	0.92	0.91	0.92	782
LSTM	0.68	0.52	0.59	441
GRU	0.71	0.53	0.61	463
DR-GCN	0.81	0.72	0.76	623
TMP	0.83	0.73	0.78	628
CGE	0.87	0.85	0.86	728
ReVulDL	**0.93**	**0.92**	**0.93**	**790**

Table 2: *Top-N Accuracy*, *MAR* and *MFR* of ReVulDL and eight baselines.

Approach	Top-1	Top-3	Top-5	Top-10	Top-20	MFR	MAR
ReVulDL	**20.40%**	**44.05%**	**58.99%**	**70.38%**	**84.05%**	**4.87**	**5.70**
GCN	9.24%	21.91%	29.82%	38.67%	51.15%	16.44	21.25
Vanilla-RNN	5.81%	12.55%	16.81%	20.06%	23.95%	29.13	34.58
Peculiar	17.52%	34.17%	47.81%	60.25%	76.78%	6.92	8.13
LSTM	6.43%	13.87%	18.59%	22.17%	26.48%	28.61	32.79
GRU	6.63%	14.32%	19.18%	22.87%	28.16%	26.95	29.47
DR-GCN	11.16%	28.87%	34.57%	42.56%	59.27%	14.25	16.37
TMP	13.38%	30.44%	37.28%	46.15%	61.23%	12.83	14.16
CGE	16.34%	33.44%	44.14%	56.82%	67.44%	8.79	11.87

4.5 Vulnerability Localization Results (RQ2)

Top-N Accuracy, MAR and MFR For developers, they expect the vulnerable statements to be top-ranked and thus they are more likely to examine and locate vulnerabilities earlier [40, 66]. Thus, our experiments use *Top-N Accuracy* (*i.e.,* N=1, 3, 5, 10, 20), *MAR*, and *MFR* to evaluate the localization effectiveness of ReVulDL. Table 2 presents their distribution among eight vulnerability localization approaches.

As shown in Table 2, ReVulDL achieves promising localization results, *e.g., Top-1* is 20.40% meaning that ReVulDL can locate the vulnerable statements at the first place in 20.40% of all *Reentrancy* vulnerabilities. Specifically, ReVulDL can localize 20.40%, 44.05%, 58.99%, 70.38%, 84.05% vulnerable statements when inspecting the *Top-1*, *Top-3*, *Top-5*, *Top-10* and *Top-20* ranked statements. Furthermore, the *MAR* and *MFR* are 4.87 and 5.70 respectively. The results

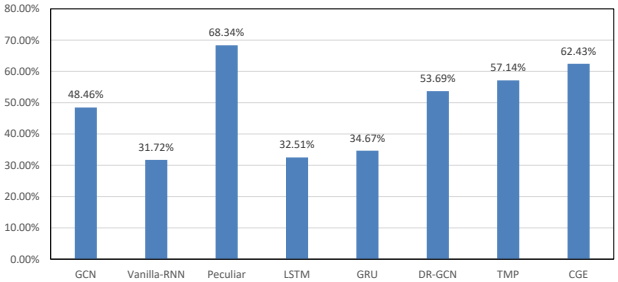

Figure 7: *RImp* distribution of ReVulDL.

Table 3: Statistical results of RQ2.

Program	2-tailed	1-tailed(right)	1-tailed(left)	Conclusion
ReVulDL vs GCN	0.03	0.90	0.02	BETTER
ReVulDL vs Vanilla-RNN	0.02	0.96	0.01	BETTER
ReVulDL vs Peculiar	0.04	0.84	0.04	BETTER
ReVulDL vs LSTM	0.02	0.99	0.01	BETTER
ReVulDL vs GRU	0.03	0.86	0.02	BETTER
ReVulDL vs DR-GCN	0.03	0.89	0.03	BETTER
ReVulDL vs TMP	0.04	0.85	0.04	BETTER
ReVulDL vs CGE	0.04	0.81	0.04	BETTER

on *Top-N Accuracy*, *MAR*, and *MFR* show that ReVulDL significantly outperforms all the seven vulnerability localization approaches.

RImp distribution For a detailed improvement, we adopt *RImp* to evaluate ReVulDL. Figure 7 represents the *RImp* distribution of ReVulDL. As shown in Figure 7, the *RImp* score is less than 100% in all approaches, meaning that ReVulDL outperforms all the other vulnerability localization approaches. The statements that need to be examined decrease ranging from 31.72% in Vanilla-RNN to 68.34% in Peculiar. It also means that ReVulDL, in comparison to the other approaches, obtains a maximum saving of 68.28% (100%-31.72%=68.28%) in Vanilla-RNN and the minimum saving is 31.66% (100%-68.34%=31.66%) in Peculiar. In summary, ReVulDL saves from 31.66% to 80.15% of the number of statements examined among all the nine baselines.

Statistical comparison To investigate whether the difference between the baselines and ReVulDL is statistically significant, we further conduct statistical comparison Wilcoxon-Signed-Rank Test [14] with a Bonferroni correction [1]. The experiments performed eight paired Wilcoxon-Signed-Rank tests between ReVulDL and each of the eight localization baselines by using *ranks* of the vulnerable statements as the pairs of measurements F(x) and G(y). Each test uses both the 2-tailed and 1-tailed checking at the σ level of 0.05. Specifically, given a vulnerability localization approach VL1, we use the list of *ranks* of ReVulDL in all vulnerable versions of

Table 4: Statistical results of RQ3.

Program	2-tailed	1-tailed(right)	1-tailed(left)	Conclusion
ReVulDL-DFR vs ReVulDL-CFR	0.03	0.86	0.02	BETTER
ReVulDL-DFR vs ReVulDL-CFRDFR	0.78	0.39	0.61	SIMILAR

all smart contracts as the list of measurements of F(x), while the list of measurements of G(y) is the list of *ranks* of VL1 in all vulnerable versions of all smart contracts. Hence, in the 2-tailed test, ReVulDL has SIMILAR effectiveness as VL1 when H_0 is accepted at the significant level of 0.05. And in the 1-tailed test (right), ReVulDL has WORSE effectiveness than VL1 when H_1 is accepted at the significant level of 0.05. Finally, in the 1-tailed test (left), VL1 using ReVulDL has BETTER effectiveness than VL1 when H_1 is accepted at the significant level of 0.05.

Table 3 shows the statistical results between ReVulDL and each baseline. We can observe that ReVulDL obtains all *BETTER* results, showing that ReVulDL is statistically significantly better than all the seven baselines in locating vulnerable statements.

4.6 Ablation Experiments (RQ3)

Graph-based pre-trained model ReVulDL utilizes a graph-based pre-trained model to perform the two-phase tasks (*i.e.,* detection and localization). It is thus interesting to investigate the contribution of this model. We normalize the parameters of the pre-trained model and retrain it with the original inputs. During the training process, the loss rate of the model decreased insignificantly and consequently the model can not converge with no vulnerable contract detected. Thus, we conclude that the graph-based pre-trained model plays an irreplaceable role in vulnerability detection and localization.

Propagation chains ReVulDL constructs the propagation chains using data flow relationships (denoted as DFR) to drive the learning process. It is thus interesting to see the contribution of our propagation chains, *i.e.,* what is the effect by constructing propagation chains using other types of relationships. There are two other major types of propagation chains: propagation chains using control flow relationships (denoted as CFR) and the ones combining control flow relationships with data flow relationships (denoted as CFRDFR). We use CFR and CFRDFR for ReVulDL respectively, and compared them with the original ReVulDL using data flow relationships. Table 4 shows the statistical comparison results using Wilcoxon-Signed-Rank Test. Each test uses the list of *ranks* of the vulnerable statements of ReVulDL as F(x) and the list of *ranks* of each baseline as G(y), checking at the σ level of 0.05. ReVulDL-DFR (*i.e.,* the original ReVulDL) obtains BETTER result on ReVulDL-CFR, meaning that the propagation chains using data flow relationships are more effective than control flow relationships in vulnerability detection and localization. ReVulDL-DFR acquires SIMILAR result on ReVulDL-CFRDFR, showing that even if we combine two types of relationships, just using data flow relationships is comparable to the combination. In addition, the combination of two types of relationships increases the learning cost, and thus ReVulDL (CFRDFR) does not improve the efficiency in comparison to ReVulDL (DFR). Thus, we conclude that propagation chains of the original ReVulDL play a vital role in *reentrancy* vulnerability detection and localization.

Table 5: *Timestamp* detection performance comparison of ReVulDL and the 13 baselines in terms of Recall, Precision and F1 score.

Approach	Timestamp			
	Recall	Precision	F1	Number
Manticore	0.50	0.57	0.50	11
Osiris	0.51	0.53	0.52	81
Oyente	0.52	0.55	0.53	107
Securify	0.55	0.53	0.53	683
Slither	0.78	0.82	0.84	1296
Smartcheck	0.50	0.74	0.51	22
GCN	0.76	0.68	0.72	1643
Vanilla-RNN	0.45	0.52	0.46	1256
LSTM	0.59	0.50	0.54	1224
GRU	0.59	0.49	0.54	1089
DR-GCN	0.79	0.71	0.75	1698
TMP	0.84	0.75	0.79	1785
CGE	0.88	0.87	0.88	1894
ReVulDL	**0.91**	**0.88**	**0.90**	**1996**

Table 6: *Top-N Accuracy*, *MAR* and *MFR* of ReVulDL and seven baselines in locating *timestamp* vulnerability.

Comparison	Top-1	Top-3	Top-5	Top-10	Top-20	MFR	MAR
ReVulDL	**17.25%**	**33.17%**	**42.83%**	**55.46%**	**61.47%**	**12.48**	**21.32**
GCN	6.53%	16.48%	21.65%	29.42%	38.19%	22.17	31.79
Vanilla-RNN	4.12%	8.72%	11.91%	15.34%	17.67%	34.25	42.68
LSTM	5.28%	9.14%	13.27%	18.51%	24.93%	29.61	35.79
GRU	5.94%	11.27%	15.35%	19.46%	27.26%	28.87	33.68
DR-GCN	7.26%	21.28%	25.72%	35.15%	44.67%	21.38	29.27
TMP	10.24%	25.56%	28.19%	43.64%	49.47%	19.35	26.61
CGE	14.16%	27.58%	35.04%	51.19%	57.33%	16.44	24.32

Table 7: Comparison of ReVulDL and Sereum.

Approach	Recall	Precision	F1	Number
ReVulDL	0.89	0.87	0.88	73
Sereum	0.62	0.57	0.59	49

5 DISCUSSION

5.1 Can ReVulDL be generalized to detect and locate other types of smart contract vulnerabilities?

It is natural to raise a question whether ReVulDL can be generalized to detect and locate other types of smart contract vulnerabilities. To evaluate this potential, we apply ReVulDL for another representative type of smart contract vulnerability (*i.e., timestamp* vulnerability [18]), and compare ReVulDL with those baselines which can work on *timestamp* vulnerabilities. Our experiments also use the dataset *SmartBugs Wild Dataset* [25] and the same parameter settings in Section 4.2.

Table 5 and Table 6 show the *timestamp* detection performance (*i.e., Recall*, *Precision* and *F1*) and localization performance (*i.e., Top-N Accuracy*, *MAR* and *MFR*) of ReVulDL over the baselines. Table 5 and Table 6 show that ReVulDL achieves the best *timestamp* detection (*i.e.,* 0.88 *Precision*, 0.91 *Recall*, 0.90 *F1* and 1996 detected vulnerabilities) and localization (*i.e.,* 17.25% *Top-1*, 33.17% *Top-3*, 42.83% *Top-5*, 55.46% *Top-10*, 61.47% *Top-20*, 12.48 *MFR* and 21.32 *MAR*) performance. It means that ReVulDL is potential to be generalized to detect and locate other types of smart contract vulnerabilities.

5.2 What types of *reentrancy* vulnerabilities can be detected by ReVulDL?

There are four major types of *reentrancy* vulnerabilities: same-function *reentrancy*, cross-function *reentrancy*, delegated *reentrancy* and create-based *reentrancy* [54, 55]. Same-function *reentrancy* causes control flow to an external contract with a fallback function, and then updates the state afterward in a single function, which may cause the state of the contract to be incomplete when flow control is transferred. In comparison to same-function *reentrancy*, cross-function *reentrancy* is a similar attack when the same contract is re-entered in a different function. Delegated *reentrancy* performs a *reentrancy* hiding in a *DELETEGATECALL* or *CALLCODE* instruction. Create-based *reentrancy* denotes that a newly created contract may issue further calls in its constructor to a malicious contract. As compared with the first two types of *reentrancy*, the delegated *reentrancy* and create-based *reentrancy* require the dynamic information with transactions to be revealed. Thus, without dynamic execution information of smart contracts, the latter two types of *reentrancy* are out of the scope of static approaches including ReVulDL, and can be mainly detected by dynamic approaches.

To evaluate the effectiveness of ReVulDL in the first two types of *reentrancy*, we further compare ReVulDL with the recent promising dynamic approach Sereum [54]. Since Sereum requires the dynamic execution information of smart contracts with many transactions, its process is much time-consuming. Due to the high cost, we randomly select 500 smart contracts from the test dataset of the experiments to conduct the comparison, where 80 are vulnerable contracts and 420 are non-vulnerable ones, to interact with these contracts under test. Table 7 illustrates the comparison of ReVulDL and Sereum, where ReVulDL acquires a higher *Recall*, *Precision*, *F1* and the number of detected vulnerabilities over Sereum, showing that our approach outperforms Sereum.

6 THREATS TO VALIDITY

Threats to internal validity Our implementation of baselines and ReVulDL may potentially include bugs. To mitigate the threats, our five team members carefully implement them according to the publicly available source code and the previous studies, and then test those approaches by hand-made test cases for their correctness.

Threats to external validity Our experiments use the large-scale popular dataset *SmartBugs Wild Dataset*, all from real-life development, However, the experimental results may not apply to all cases since there are still many unknown and complicated factors in realistic development that could affect the experimental results. Thus, it is worthwhile to conduct experiments on more smart contracts to further strengthen the experimental results.

Threats to construct validity We adopt seven widely used metrics to evaluate the effectiveness. According to the extensive use of the measurement, the threat is acceptably mitigated.

7 RELATED WORK

This section surveys closely related work on contract vulnerability detection.

There are conventional smart contract vulnerability detection techniques. Bhargavan *et al.* [4] provide a verification system that inputs Ethereum virtual machine (EVM) bytecode and Solidity code. Grishchenko *et al.* [27] utilize a F* framework and Hildenbrandt *et al.* [31] use a K framework to define EVM formal semantics. Brenner *et al.* [7] detects vulnerabilities in smart contracts using the Isabelle/HOL tool. Jiang *et al.* [35] propose ContractFuzzer using fuzzing technology with dynamic executions to detect vulnerabilities. Rodler *et al.* [54] develops Sereum monitoring runtime data flows during smart contract execution using taint analysis. Some researchers leverage symbolic execution (*e.g.,* Oyente [45], Osiris [60], Mythril [48] and Manticore [47]), program analysis (*e.g.,* Smartcheck[59] and Slither[23]), and formal verification (*e.g.,* ZEUS [36] and Securify [62]) to detect smart contract vulnerabilities. The above conventional approaches are fundamentally based on several expert-defined rules and patterns manually summarized in advance. However, the manually summarized rules and patterns may suffer from the inherent risk of being error-prone and the tricks used by crafty attackers who may surpass these already known rules.

To overcome these limitations, many researchers recently have applied deep learning to smart contract vulnerability detection, *e.g.,* SmartEmbed [26], DR-GCN [72], TMP [72], CGE [43], Qian *et al.* [51] and Tann *et al.* [58]. SmartEmbed [26] utilizes similarity calculation to detect vulnerabilities in smart contracts. DR-GCN, TMP [72] and CGE [43] use graph neural network to learn both syntactic and semantic features of smart contracts and detect vulnerabilities. Qian *et al.* [51] customize LSTM based networks to sequentially process smart contract source code while Tann *et al.* [58] utilize a sequential model to analyze the Ethereum operation code.

Despite delivering promising results, the above deep learning based approaches focus on detecting whether a smart contract is vulnerable or not whereas their results on locating vulnerable statements are still unsatisfactory. Thus, we propose a deep learning based two-phase approach ReVulDL, trying to study vulnerability detection and localization as a debugging pipeline for further improvement.

8 CONCLUSION AND FUTURE WORK

In this paper, we propose a deep learning based two-phase approach ReVulDL to detect and locate *reentrancy* vulnerability in smart contracts. For detection phase, ReVulDL customizes a graph-based pre-trained model using propagation chains to learn a model to detect whether a smart contract is vulnerable or not for *reentrancy* vulnerability; for localization phase, ReVulDL utilizes post-hoc interpretable machine learning to learn and evaluate the suspiciousness of each statement of being vulnerable. Our experimental results show that ReVulDL is statistically significantly effective in detecting and locating *reentrancy* vulnerability.

In future, we plan to explore more on interpretable machine learning and extend our approach to other types of vulnerabilities (*e.g., infinite loop* vulnerability [18]).

ACKNOWLEDGMENTS

This work is partially supported by the National Key Research and Development Project of China (No. 2020YFB1711900), the Natural Science Foundation of Chongqing (No. cstc2021jcyj-msxmX0538), the National Nature Science Foundation of China (No.62002034), and the Major Key Project of PCL.

REFERENCES

[1] Hervé Abdi. 2007. The Bonferonni and Šidák Corrections for Multiple Comparisons. *Encyclopedia of measurement and statistics* 3 (2007), 103–107.

[2] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. 2017. Learning to represent programs with graphs. *arXiv preprint arXiv:1711.00740* (2017).

[3] Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. 2018. code2seq: Generating Sequences from Structured Representations of Code. In *Proceedings of the International Conference on Learning Representations*.

[4] Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Anitha Gollamudi, Georges Gonthier, Nadim Kobeissi, Natalia Kulatova, Aseem Rastogi, Thomas Sibut-Pinote, Nikhil Swamy, et al. 2016. Formal verification of smart contracts: Short paper. In *Proceedings of the 2016 ACM workshop on programming languages and analysis for security*. 91–96.

[5] Umang Bhatt, Alice Xiang, Shubham Sharma, Adrian Weller, Ankur Taly, Yunhan Jia, Joydeep Ghosh, Ruchir Puri, José MF Moura, and Peter Eckersley. 2020. Explainable machine learning in deployment. In *Proceedings of the 2020 conference on fairness, accountability, and transparency*. 648–657.

[6] B. Boehmke and B. Greenwell. 2019. *Interpretable Machine Learning*. Hands-On Machine Learning with R.

[7] M. Brenner, K. Rohloff, J. Bonneau, A. Miller, Pya Ryan, V. Teague, A. Bracciali, M. Sala, F. Pintore, and M. Jakobsson. 2017. [Defining the Ethereum Virtual Machine for Interactive Theorem Provers. 10.1007/978-3-319-70278-0, Chapter 33 (2017), 520–535.

[8] Luca Buratti, Saurabh Pujar, Mihaela Bornea, Scott McCarley, Yunhui Zheng, Gaetano Rossiello, Alessandro Morari, Jim Laredo, Veronika Thost, Yufan Zhuang, et al. 2020. Exploring software naturalness through neural language models. *arXiv preprint arXiv:2006.12641* (2020).

[9] Vitalik Buterin et al. 2013. Ethereum white paper. *GitHub repository* 1 (2013), 22–23.

[10] David Chandler. 1987. Introduction to modern statistical. *Mechanics. Oxford University Press, Oxford, UK* 5 (1987).

[11] Angelos Chatzimparmpas, Rafael Messias Martins, Ilir Jusufi, Kostiantyn Kucher, Fabrice Rossi, and Andreas Kerren. 2020. The state of the art in enhancing trust in machine learning models with the use of visualizations. In *Computer Graphics Forum*, Vol. 39. Wiley Online Library, 713–756.

[12] Jiachi Chen, Xin Xia, David Lo, John Grundy, Xiapu Luo, and Ting Chen. 2022. Defining Smart Contract Defects on Ethereum. *IEEE Transactions on Software Engineering (TSE)* 48, 1 (2022), 327–345. https://doi.org/10.1109/TSE.2020.2989002

[13] Weili Chen, Mingjie Ma, Yongjian Ye, Zibin Zheng, and Yuren Zhou. 2018. IoT service based on jointcloud blockchain: The case study of smart traveling. In *Proceedings of the 2018 IEEE Symposium on service-oriented system engineering (SOSE)*. IEEE, 216–221.

[14] Gregory W. Corder and Dale I. Foreman. 2010. *Nonparametric Statistics for Non-Statisticians: A Step-by-Step Approach*. Vol. 78. International Statistical Review. 451–452 pages.

[15] Chris Dannen. 2017. *Introducing Ethereum and solidity*. Vol. 318. Springer.

[16] Vidroha Debroy, W. Eric Wong, Xiaofeng Xu, and Byoungju Choi. 2010. A Grouping-Based Strategy to Improve the Effectiveness of Fault Localization Techniques. In *Proceedings of the International Conference on Quality Software (QSIC)*. 13–22.

[17] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805* (2018).

[18] Wesley Dingman, Aviel Cohen, Nick Ferrara, Adam Lynch, Patrick Jasinski, Paul E Black, and Lin Deng. 2019. Classification of smart contract bugs using the nist bugs framework. In *Proceedings of the 17th International Conference on Software Engineering Research, Management and Applications (SERA)*. IEEE, 116–123.

[19] Finale Doshi-Velez and Been Kim. 2017. Towards a rigorous science of interpretable machine learning. *arXiv preprint arXiv:1702.08608* (2017).

[20] Finale Doshi-Velez, Mason Kortz, Ryan Budish, Chris Bavitz, Sam Gershman, David O'Brien, Kate Scott, Stuart Schieber, James Waldo, David Weinberger, et al. 2017. Accountability of AI under the law: The role of explanation. *arXiv preprint arXiv:1711.01134* (2017).

[21] M. Du, N. Liu, and X. Hu. 2018. Techniques for Interpretable Machine Learning. (2018).

[22] Mengnan Du, Ninghao Liu, and Xia Hu. 2019. Techniques for interpretable machine learning. *Commun. ACM* 63, 1 (2019), 68–77.

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

[24] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. 2020. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155* (2020).

[25] JoÃ£o F. Ferreira, Pedro Cruz, Thomas Durieux, and Rui Abreu. 2020. SmartBugs: A Framework to Analyze Solidity Smart Contracts. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 1349–1352.

[26] Zhipeng Gao, Vinoj Jayasundara, Lingxiao Jiang, Xin Xia, David Lo, and John Grundy. 2019. Smartembed: A tool for clone and bug detection in smart contracts through structural code embedding. In *Proceedings of the 2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 394–397.

[27] Ilya Grishchenko, Matteo Maffei, and Clara Schneidewind. 2018. A semantic framework for the security analysis of ethereum smart contracts. In *Proceedings of the International Conference on Principles of Security and Trust*. Springer, 243–269.

[28] Anbang Guo, Xiaoguang Mao, Deheng Yang, and Shangwen Wang. 2018. An empirical study on the effect of dynamic slicing on automated program repair efficiency. In *Proceedings of the 2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 554–558.

[29] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, et al. 2020. Graphcodebert: Pre-training code representations with data flow. *arXiv preprint arXiv:2009.08366* (2020).

[30] Vincent J Hellendoorn, Charles Sutton, Rishabh Singh, Petros Maniatis, and David Bieber. 2019. Global relational models of source code. In *International conference on learning representations*.

[31] Everett Hildenbrandt, Manasvi Saxena, Nishant Rodrigues, Xiaoran Zhu, Philip Daian, Dwight Guth, Brandon Moore, Daejun Park, Yi Zhang, Andrei Stefanescu, et al. 2018. Kevm: A complete formal semantics of the ethereum virtual machine. In *Proceedings of the 31st Computer Security Foundations Symposium (CSF)*. IEEE, 204–217.

[32] Joran Honig. 2022. tree-sitter-solidity. https://github.com/JoranHonig/tree-sitter-solidity.

[33] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2019. Codesearchnet challenge: Evaluating the state of semantic code search. *arXiv preprint arXiv:1909.09436* (2019).

[34] Suzana Ilić, Edison Marrese-Taylor, Jorge A Balazs, and Yutaka Matsuo. 2018. Deep contextualized word representations for detecting sarcasm and irony. *arXiv preprint arXiv:1809.09795* (2018).

[35] Bo Jiang, Ye Liu, and WK Chan. 2018. Contractfuzzer: Fuzzing smart contracts for vulnerability detection. In *Proceedings of the 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 259–269.

[36] Sukrit Kalra, Seep Goel, Mohan Dhawan, and Subodh Sharma. 2018. ZEUS: Analyzing Safety of Smart Contracts.. In *Ndss*. 1–12.

[37] Aditya Kanade, Petros Maniatis, Gogul Balakrishnan, and Kensen Shi. 2019. Pre-trained contextual embedding of source code. (2019).

[38] Rafael-Michael Karampatsis and Charles Sutton. 2020. Scelmo: Source code embeddings from language models. *arXiv preprint arXiv:2004.13214* (2020).

[39] Seohyun Kim, Jinman Zhao, Yuchi Tian, and Satish Chandra. 2021. Code prediction by feeding trees to transformers. In *Proceedings of the 43rd International Conference on Software Engineering (ICSE)*. IEEE, 150–162.

[40] Pavneet Singh Kochhar, Xin Xia, David Lo, and Shanping Li. 2016. Practitioners' expectations on automated fault localization. In *Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA)*. 165–176.

[41] Xia Li, Wei Li, Yuqun Zhang, and Lingming Zhang. 2019. DeepFL: Integrating multiple fault diagnosis dimensions for deep fault localization. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. 169–180.

[42] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692* (2019).

[43] Zhenguang Liu, Peng Qian, Xiaoyang Wang, Yuan Zhuang, Lin Qiu, and Xun Wang. 2021. Combining graph neural networks with expert knowledge for smart contract vulnerability detection. *IEEE Transactions on Knowledge and Data Engineering* (2021).

[44] Ting-gan Lua. 2022. tree-sitter. https://tree-sitter.github.io/tree-sitter/.

[45] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. 2016. Making smart contracts smarter. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*. 254–269.

[46] Christoph Molnar. 2020. *Interpretable machine learning*. Lulu. com.

[47] Mark Mossberg, Felipe Manzano, Eric Hennenfent, Alex Groce, Gustavo Grieco, Josselin Feist, Trent Brunson, and Artem Dinaburg. 2019. 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)*. IEEE, 1186–1189.

[48] B Mueller. 2017. Mythril-Reversing and bug hunting framework for the Ethereum blockchain.

[49] Satoshi Nakamoto. 2008. Bitcoin: A peer-to-peer electronic cash system. *Decentralized Business Review* (2008), 21260.

[50] Chris Parnin and Alessandro Orso. 2011. Are automated debugging techniques actually helping programmers?. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*. 199–209.

[51] Peng Qian, Zhenguang Liu, Qinming He, Roger Zimmermann, and Xun Wang. 2020. Towards automated reentrancy detection for smart contracts based on sequential models. *IEEE Access* 8 (2020), 19685–19695.

[52] Alec Radford, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever. 2018. Improving language understanding by generative pre-training. (2018).

[53] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. 2019. Exploring the limits of transfer learning with a unified text-to-text transformer. *arXiv preprint arXiv:1910.10683* (2019).

[54] Michael Rodler, Wenting Li, Ghassan O Karame, and Lucas Davi. 2018. Sereum: Protecting existing smart contracts against re-entrancy attacks. *arXiv preprint arXiv:1812.05934* (2018).

[55] Noama Fatima Samreen and Manar H Alalfi. 2020. Reentrancy vulnerability identification in Ethereum smart contracts. In *2020 IEEE International Workshop on Blockchain Oriented Software Engineering (IWBOSE)*. IEEE, 22–29.

[56] David Siegel. 2016. Understanding the DAO Attack. *Retrieved June* 13 (2016), 2018.

[57] Alexey Svyatkovskiy, Shao Kun Deng, Shengyu Fu, and Neel Sundaresan. 2020. 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*. 1433–1443.

[58] A Tann, Xing Jie Han, Sourav Sen Gupta, and Yew-Soon Ong. 2018. Towards safer smart contracts: A sequence learning approach to detecting vulnerabilities. *arXiv preprint arXiv:1811.06632* (2018), 1371–1385.

[59] Sergei Tikhomirov, Ekaterina Voskresenskaya, Ivan Ivanitskiy, Ramil Takhaviev, Evgeny Marchenko, and Yaroslav Alexandrov. 2018. Smartcheck: Static analysis of ethereum smart contracts. In *Proceedings of the 1st International Workshop on Emerging Trends in Software Engineering for Blockchain*. 9–16.

[60] Christof Ferreira Torres, Julian Schütte, and Radu State. 2018. Osiris: Hunting for integer bugs in ethereum smart contracts. In *Proceedings of the 34th Annual Computer Security Applications Conference*. 664–676.

[61] Christof Ferreira Torres, Mathis Steichen, et al. 2019. The art of the scam: Demystifying honeypots in ethereum smart contracts. In *Proceedings of the 28th USENIX Security Symposium (USENIX Security 19)*. 1591–1607.

[62] Petar Tsankov, Andrei Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Buenzli, and Martin Vechev. 2018. Securify: Practical security analysis of smart contracts. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 67–82.

[63] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems* 30 (2017).

[64] Yaron Velner, Jason Teutsch, and Loi Luu. 2017. Smart contracts make bitcoin mining pools vulnerable. In *Proceedings of the International Conference on Financial Cryptography and Data Security*. Springer, 298–316.

[65] Hongjun Wu, Zhuo Zhang, Shangwen Wang, Yan Lei, Bo Lin, Yihao Qin, Haoyu Zhang, and Xiaoguang Mao. 2021. Peculiar: Smart Contract Vulnerability Detection Based on Crucial Data Flow Graph and Pre-training Techniques. In *Proceedings of the 32nd International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 378–389. https://doi.org/10.1109/ISSRE52982.2021.00047

[66] X. Xia, L. Bao, D. Lo, and S. Li. 2016. "Automated Debugging Considered Harmful" Considered Harmful: A User Study Revisiting the Usefulness of Spectra-Based Fault Localization Techniques with Professionals Using Real Bugs from Large Systems. In *Proceedings of the IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 267–278. https://doi.org/10.1109/ICSME.2016.67

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

[68] Meng Yan, Xin Xia, Yuanrui Fan, Ahmed E. Hassan, David Lo, and Shanping Li. 2022. Just-In-Time Defect Identification and Localization: A Two-Phase Framework. *IEEE Transactions on Software Engineering* 48, 1 (2022), 82–101. https://doi.org/10.1109/TSE.2020.2978819

[69] R. Ying, D. Bourgeois, J. You, M. Zitnik, and J. Leskovec. 2019. GNNExplainer: Generating Explanations for Graph Neural Networks. *Advances in neural information processing systems* 32 (2019), 9240–9251.

[70] Zhitao Ying, Dylan Bourgeois, Jiaxuan You, Marinka Zitnik, and Jure Leskovec. 2019. Gnnexplainer: Generating explanations for graph neural networks. *Advances in neural information processing systems* 32 (2019).

[71] Zhuo Zhang, Yan Lei, Xiaoguang Mao, and Panpan Li. 2019. CNN-FL: An Effective Approach for Localizing Faults using Convolutional Neural Networks. In *Proceedings of the 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 445–455.

[72] Yuan Zhuang, Zhenguang Liu, Peng Qian, Qi Liu, Xiang Wang, and Qinming He. 2021. Smart contract vulnerability detection using graph neural networks. In *Proceedings of the Twenty-Ninth International Conference on International Joint Conferences on Artificial Intelligence (IJCAI)*. 3283–3290.