A Graph Neural Network-based Code Recommendation Method for Smart Contract Development

Xiuwen Tang, Jiazhen Gan
School of Computer Science and Engineering
Sun Yat-Sen University
Guangzhou, China
tangxw23@mail2.sysu.edu.cn, ganjzh3@mail2.sysu.edu.cn

Zigui Jiang*
School of Software Engineering
Sun Yat-Sen University
Zhuhai, China
jiangzg3@mail.sysu.edu.cn

2022 International Conference on Service Science (ICSS) | 978-1-6654-9861-6/22/$31.00 ©2022 IEEE | DOI: 10.1109/ICSS55994.2022.00046

***Abstract*—Smart contracts can be considered as a service in the blockchain system and have been applied in many fields, covering financial products, online games, real estate, transportation and logistics. However, smart contract technology is still in its infancy. Development task is facing many difficulties and challenges, thus providing a set of new or improved development aids for the smart contract ecosystem is an urgent problem that needs to be solved. This paper proposes a smart contract code recommendation method based on graph neural network, which aims to facilitate the development of smart contracts and help developers realize smart contracts faster and more securely. Experimental results show that this method is better than the existing model of smart contract code recommendation in terms of accuracy.**

***Keywords*-blockchain, smart contract, software development, code recommendation**

I. Introduction

Known as blockchain 2.0, Ethereum is the largest public blockchain platform at present, attracting wide attention from industry and academia. The biggest feature of Ethereum is that providing a turing-complete platform to support smart contracts. A smart contract is essentially a piece of software code that can be triggered and executed according to pre-specified conditions which can be customized by users within the blockchain for transaction behavior between users. Therefore, smart contracts can be considered as a special form of software service in blockchain systems.

However, smart contract technology is in its infancy, the ecosystem development is not yet perfect, professional technology is lacking and talent reserves are insuffcient. More importantly, due to the characteristics of smart contracts, smart contract development is very different from traditional software development and faces new problems and challenges.

The sources of problems and challenges can be divided into two categories, the first of which comes from the smart contract feature: there are costs in the deployment and execution of smart contracts. The cost of consumption depends on the developer's design of the smart contract, requiring the developer to have a good design pattern foundation and code optimization awareness. More importantly, smart contracts are immutable once deployed. Any functional defects or vulnerabilities within the contracts can lead to catastrophic losses, such as the DAO attack in 2016[1] and Parity attack in 2017.

The second category comes from smart contract developers: almost all smart contract engineers are currently transformed from software engineers, lacking a comprehensive and deeper understanding of blockchain systems. Although there are similarities between smart contracts and traditional software, due to the characteristics of smart contracts, there is a development threshold. What's more, most development engineers are not deeply involved in security issues. In the blockchain ecosystem, open source is a basic unspoken rule. When the code is all exposed to the sun, security issues are particularly important. Implementing security features in code that are not directly related to basic functionality can be more expensive to perform, but low-quality contracts can lead to gaps in the vague definition of rights and obligations. Developing smart contracts that balance performance and security is a tough challenge for development engineers.

Research by Bosu et al. [1] shows that the smart contract ecosystem requires a set of new or improved tools, such as integrated development environments for smart contract development tasks. The implementation of a smart contract development environment is critical to the building and diffusion of blockchain-oriented software engineering knowledge, which can simplify the development of smart contracts [2]. Code recommendation is considered to be one of the most useful features in modern integrated development environments. It can automatically complete the next token according to the previous part of the program, reducing the amount of input required by developers, eliminating errors or vulnerabilities caused by character input and speeding up the development speed. Research by Murphy et al. [3] shows that Eclipse IDE users use Eclipse code recommendations just as much as they use common editing commands such as copy and paste.

[1]https://www.coindesk.com/learn/2016/06/25/understanding-the-dao-attack/

978-1-6654-9861-6/22/$31.00 ©2022 IEEE
DOI 10.1109/ICSS55994.2022.00046

Therefore this paper proposes a code recommendation method based on graph neural network for smart contracts, which is one of the most important features of integrated development environment for developers. The purpose is to improve the code recommendation accuracy and help developers to implement smart contracts more quickly and safely. The main contributions of this paper are summarized as follows:

- First, we provide a high-quality dataset for studying smart contract code recommendation. Considering that the high degree of code cloning of Ethereum smart contracts will affect the model training effect and accuracy, the source code is deduplication processed to solve the data leakage problem caused by a large number of duplicate code to the model. For smart contract security issues, vulnerability detection is done on the source code to improve code security.
- Secondly, we use the abstract syntax tree as the node, the data flow information and the control flow information as the edge to build the program representation diagram, giving the program representation diagram rich features information and strengthening the language model's ability to understand semantics. Experimental results show that the strategy used to construct the program representation can help the language model improve the recommendation accuracy.
- Finally, we use a language model based on graph neural networks, adding a multi-head attention mechanism to blend the advantages of structured and unstructured code models to help the model balance global and local information. Experimental results show that this method is better than the existing model of smart contract code recommendation in terms of accuracy.

The rest of this paper is organized as follows. Section II introduces the related works. Section III presents the proposed smart contract code recommendation method and Section IV conducts experiments on smart contracts from Ethereum to evaluate our proposed code recommendation method. Finally, the paper is concluded in Section V.

II. RELATED WORK

In the field of software engineering, how to improve the efficiency and quality of software development has always been a matter of great concern. Many researchers have solved it by proposing software engineering development methods and technical means. The characteristics of smart contracts suggest that smart contract development is significantly different from traditional software development, but few software engineering researchers have focused on the former. As a result, only a small number of studies have proposed smart contract development tuning tools that help developers from different angles and granularities.

The first type is auxiliary tools or resources in the unit of the entire contract. Huang et al. [4] obtain similar contracts through clustering analysis. They extract different codes from them, then evaluate whether similar smart contracts and current evolving smart contracts involve the same code changes. If any, recommend the similar smart contracts to guide the update of the current smart contract in the next version. Support development resources are smart contract templates or secure smart contract collections. These templates and collections are usually some code that has been proven on the blockchain. ZeppelinSolutions is a smart contract review service provider that provides solutions to the difficulties of smart contract development. They use their experience in contract review to organize smart contract best practices into OpenZeppelin [5], forming a smart contract code base for developers to use and extend to maximize the security of smart contracts in less time.

The second type is the auxiliary tools in the unit of contract code, typically there is code recommendation. At present, the research scope of smart contract code recommendation is small, mainly for industrial practice. Traditional integrated development environments (IDEs) support smart contract development environments by installing extensions, such as Intellij IDEA[2] developed by JetBrains, Eclipse[3] and Atom[4]. The code recommendation function provided by these plugins and Ethereum official recommended online IDE Remix[5] only support fuzzy matching, which relies on static analysis of libraries and index files. The recommended words tend to be alphabetically ordered, resulting in low accuracy and increasing time for developers to select candidate words. Ren et al. [6] [7]use Bidirectional Long Short-Term Memory (BiLSTM) as a language model and enhance the security of the input contract data. Compared with Remix, BiLSTM improves the accuracy of code recommendation.

In the first type of study, the method of contract recommendation is limited by the similarity of contracts and the actual availability of the recommended contract is questionable. The contract template approach requires a lot of manual work and cannot handle the flexible needs of developers. The second type of auxiliary tools have the problems of weak smart contract pertinence and insufficient accuracy of smart contract code recommendation. Aiming at the problems of the second type of auxiliary tools, this paper proposes a smart contract code recommendation model based on graph neural network to solve the problem of insufficient pertinence and accuracy.

[2]https://plugins.jetbrains.com/plugin/9475-solidity/
[3]https://marketplace.eclipse.org/content/yakindu-solidity-tools
[4]https://atom.io/packages/linter-solidity
[5]https://remix.ethereum.org/

III. SMART CONTRACT CODE RECOMMENDATION

A. Program Representation

Any programming language has a displayed context-free syntax, which can be used to parse source code into an abstract syntax tree(AST). The data structure of the AST is a tree, where each non-leaf node has information about a non-terminal structure in the context-free syntax, such as for and if statements. Each leaf node corresponds to a terminal structure information in context-free syntax, such as variable names and operators. Fig. 1 is an example of an AST translated from source code 1.

Listing 1: smart contract source code sample

```
pragma solidity ^0.5.0;
contract Test {
    function addOne(uint b) public view
    returns(uint){
        uint a = 1;
        uint result = a + b;
        return result;
    }
}
```

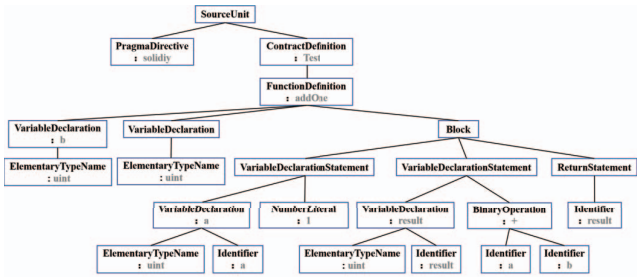


Figure 1: Abstract syntax tree sample

The method based on abstract syntax tree representation can extract and utilize the grammatical rules and structural information in source code. In order to make full use of the syntax and semantic information in the source code and reduce the loss of precision caused by redundant statements such as comments and blank lines in the code, we transform source code into abstract syntax tree by Solidity-parser-antlr[6].Then we construct a program representation diagram using nodes in the abstract syntax tree.

To better model relationships between codes, the design of adjacency relationships between nodes is important prior information and interaction constraints. We abstract the control flow and data flow information between nodes as the edges between nodes based on research [8]:

1) *NextNode*: If two nodes are adjacent in the depth-first traversal output of the abstract syntax tree, a NEXTNODE edge is added between the two nodes.
2) *NextToken*: After removing nodes with empty node values, a NEXTTOKEN edge is added between the two nodes if they are adjacent in the depth-first traversal output of the abstract syntax tree.
3) *ConditionTrue* and *ConditionFalse*: A CONDITIONTRUE edge is added between the conditional subtree root node and the true branching subtree root node of the conditional branching structure. A CONDITIONFALSE edge is added between the conditional subtree root node and the false branching root node. In Solidity, both the trinomial operator and the conditional branch statement have such a structure.
4) *WhileExec*:A WHILEEXEC edge is added between the root node of the loop condition subtree of the while loop structure and the root node of the loop body subtree.
5) *ForExec* and *ForNext*: A FOREXEC edge is added between the loop condition subtree of the for loop structure and the root of the loop body subtree. A FORNEXT edge is added between the root node of the loop body subtree and the root node of the update statement.
6) *LastRead*: A LASTREAD edge is added between the current variable node and the variable node that was last seen and was read.
7) *LastWrite*: A LASTWRITE edge is added between the current variable node and the variable node that last appeared and was written .
8) *NextUsed*: A NEXTUSED edge is added between the current variable node and its next occurrence.

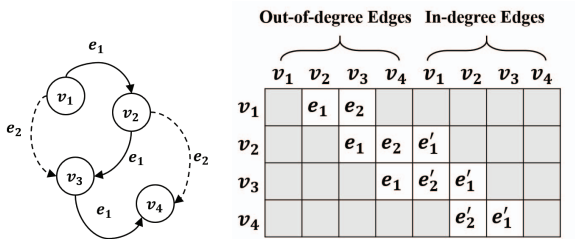
B. Code Model

The key idea of graph neural networks is to embed node representations from local domains through domain aggregation. Gated graph neural networks(GGNN) use gated recurrent units (GRU) as recursive functions to reduce recursion to a fixed number of steps [9]. Moreover, GGNN introduces a learnable parameter of w for different types of edges, thereby can processes various graphs that traditional graph neural networks cannot handle. GGNN is more suitable for data semantics and graph structure, so we use gated graph neural networks as the basic model.

Let $G = (V, E, A)$ be the program representation diagram, as shown in Fig. 2a. V is a set of nodes and E is a set of directed edges of different edge classes types. A is the $N \times N$ adjacency matrix of diagram G, which holds the edge data between nodes, as shown in Fig. 2b, where N represents the number of nodes.

The learning goal of graph neural networks is to obtain graph-aware hidden states for each node, mainly by updating and iterating on all nodes' hidden states. Setting h_v indicates the hidden state of node v, x_v indicates the characteristics of node v. Before the iterative update of the whole graph, the hidden state h_v of each node are initialized with the node

[6]https://github.com/solidity-parser/parser



(a) code presentation (b) adjacency matrix

Figure 2: GGNN

feature x_v. When the node feature dimension is less than the hidden state dimension, add 0 after that, as shown in Equation (1):

$$h_v{}^{(1)} = [x_v{}^T, 0]^T. \tag{1}$$

Information is passed between nodes through the outgoing edges and incoming edges. Let $A_{v:}$ be the out-of-degree edge data and the in-degree edge data corresponding to the adjacency matrix A and the node v, including the activation information a_v from different edge types and directions accepted by node v. The activation information is calculated as Equation (2):

$$a_v{}^{(t)} = A_{v:}{}^T[h_1{}^{(t-1)T}, \cdots, h_{|V|}{}^{(t-1)T}]^T + b, \tag{2}$$

where $[h_1{}^{(t-1)T}, \cdots, h_{|v|}{}^{(t-1)T}]$ is a $D|V|$ dimension vector formed by the stitching of all nodes in the hidden state at t-1.

The state vector is updated within each time step t, calculated as in the Equation (3) to (6):

$$z_v{}^t = \sigma\left(W^z a_v{}^{(t)} + U^z h_v{}^{(t-1)}\right), \tag{3}$$

$$r_v{}^t = \sigma\left(W^r a_v{}^{(t)} + U^r h_v{}^{(t-1)}\right), \tag{4}$$

$$\widetilde{h_v{}^{(t)}} = tanh\left(W a_v{}^{(t)} + U\left(r_v{}^t \odot h_v{}^{(t-1)}\right)\right), \tag{5}$$

$$h_v{}^{(t)} = (1 - z_v{}^t) \odot h_v{}^{(t-1)} + z_v{}^t \odot h_v{}^{(t)}, \tag{6}$$

where $z_v{}^t$ is the update gate that controls forgotten information. $r_v{}^t$ is the reset door controlling the generation of new information. $\widetilde{h_v{}^{(t)}}$ is the newly generated information and $h_v{}^{(t)}$ is the hidden state of the node that is finally updated.

After propagation and update with a time step of T, the state vector matrix $H = [h_1{}^{(T)}, h_v{}^{(T)}, \cdots, h_{|V|}{}^{(T)}]$ for all nodes in the entire graph is obtained.

C. Attention Mechanism

To capture the importance of each node in the graph to other nodes, a multi-head attention mechanism is incorporated during the propagation phase of the node to provide a global view of the code. The multi-head attention mechanism is derived from Transformer [10]. Its core contains three element: queries (Q), key (K) and value (V). This attention mechanism is actually an addressing process that given a task-related query vector of Q, obtains the attention value by calculating the attention distribution with the key-value vector K and attaching the result to the value vector V.

For the state vector matrix H of the node obtained in the propagation stage, the multi-head attention mechanism has a total of n heads and the state vector matrix H is cut into n parts. The representations of query vectors Q_i, key vectors K_i and value vector V_i are queried in each head attention, respectively, as shown in the Equation (7), (8) and (8) :

$$Q_i = W_i{}^Q H_i, \tag{7}$$

$$K_i = W_i{}^K H_i, \tag{8}$$

$$V_i = W_i{}^Q H_i. \tag{9}$$

According to the Equation (10), the output of each head's attention is calculated to be $head_i$. The calculation result of the multi-head attention is combined to obtain H_{MHA}, as shown in the Equation (11):

$$head_i = softmax\left(\frac{Q_i k_i{}^T}{\sqrt{d_k}}\right) V_i, \tag{10}$$

$$H_{MHA} = concact\left(head_1, \cdots, head_n\right) W_O. \tag{11}$$

Finally, the multi-head attention splicing result of H_{MHA} through the feed-forward neural network gets the output of the attention layer of H_A, as shown in the Equation (12), which is used as input to the next propagation stage in the code model.

$$H_A = W_2 ReLu\left(W_1 H_{MHA} + b_1\right) + b_2, \tag{12}$$

D. Code Recommendation

Considering that the rightmost node of the abstract syntax tree has the most useful information for the next node [11], we use the state vector of the rightmost node to generate the type probability distribution and value probability distribution of the next node, corresponding to the type prediction task and the value prediction task.

As in Equations (13) and (14), the rightmost node state vector h_r is mapped to the node space, where y_t is the probability distribution of the next node to be predicted and y_v is the value probability distribution. The corresponding code words of the nodes are arranged in descending order of probability values as the predicted code list.

$$y_t = softmax\left(W_t h_r{}^t + b_1\right) + b_t, \tag{13}$$

$$y_v = softmax\left(W_v h_r{}^v + b_1\right) + b_v. \tag{14}$$

IV. EXPERIMENTAL RESULTS AND ANALYSIS

The dataset for the experiment comes from Etherscan[7], which is the most widely used blockchain browser on ethereum. It's a free site that can search for transaction block wallet addresses, smart contracts and other on-chain data.

We got a total of 232,923 open source smart contracts. There are 141,573 left after deduplication and 34,856 contracts were selected from them. After using Securify [12], Mythril [13], Smartian [14] tools for detection, 30471 secure smart contracts were obtained.

A. Comparison Methods

In order to verify the effectiveness of the proposed model, this section compares the existing methods in the smart contract code recommendation and the best traditional code recommendation methods:

- *BiLSTM* [6]:The method serializes the source code, on the basis of the bidirectional long short-term memory network forms a bidirectional attention LSTM model for code recommendation.
- *GAT* [11]:The method uses graph attention networks as a code model and designs attention blocks to capture multiple code dependencies such as order, structural information and repeating patterns.
- *GGNN* [9]:This paper proposes GGNN and many code recommendation methods are derived from this network [15].
- *GGNN-Transformer*:The method proposed in this paper. Using abstract syntax trees, control flow edges and data flow edges to build program representation diagrams. The multi-head attention mechanism of Transformer is incorporated into the GGNN basic code model.

B. Metrics

Accuracy refers to whether the currently predicted outcome matches the correct outcome, as opposed to the accuracy rate in machine learning. The calculation is as shown in Equation (15):

$$acc = \frac{\sum_{i=1}^{N} T_i^{pred}}{N}, \tag{15}$$

where the value of T_i^{pred} is taken 1 when the prediction matches the correct result and 0 when there is no match.

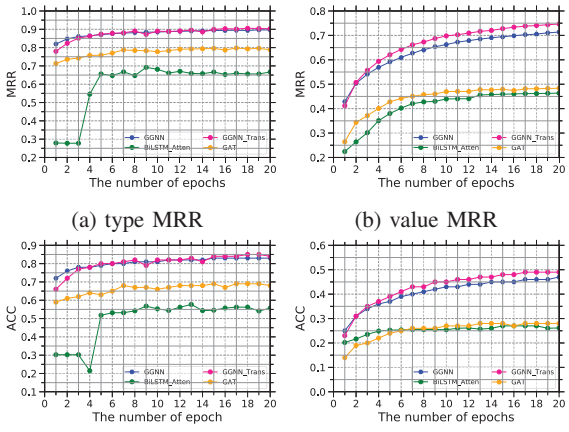
Mean reciprocal rank is an internationally common mechanism for evaluating search algorithms, that is, the first result match has a score of 1, the second result matches a score of 0.5, and so on the Nth match score is 1/n, if there is no matching sentence score of 0. The final score is the sum of all the scores. The reason why we use not only accuracy but also MRR as an evaluation index is that the accuracy rate is scored only when the correct label is at the top of the prediction list, while in the scenario of code recommendation, the use of MRR is closer to the real-world scenario of providing code recommendation to developers. In particular, for traditional MRR, we make a ranking limit that the match score after ranking K is 0. Since in a real development environment, developers usually only choose the top few of the recommended list, rather than going down the recommendation list until they find the target word. Improved MRR calculation such as Equation (16):

$$mrr = \frac{\sum_{i=1}^{N} \frac{1}{rank_i}}{N}. \tag{16}$$

[7]https://cn.etherscan.com/

C. Results and Comparison

Fig. 3 shows the experimental results of each of the four types of models. In the MRR indicator, the GGNN-based generation code recommendation method and the GGNN-Transformer method proposed in this paper have obtained good performance in both node type prediction and node value prediction tasks. Node type prediction tasks generally perform better than node value prediction tasks, mainly because the number of node types is limited, with only 75 types. The regularity of node types in the program is stronger that after giving the previous node type, the next node type will be greatly constrained, or even only the only choice. For example, the next node type of the $VariableDeclarationStatement$ node can only be the $VariableDeclaration$ type. The ACC indicator is generally lower than the MRR indicator, which is consistent with the indicator definition.



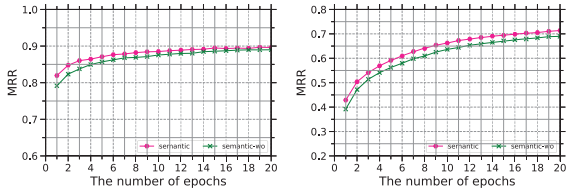
(a) type MRR (b) value MRR

(c) type ACC (d) value ACC

Figure 3: Model comprehensive evaluation

Since GGNNs introduce parameters for different types of edges, which can be learned to process various graphs that traditional graph neural networks cannot handle, the GGNN model is used to study the effects of data flow edges and control flow edges. Fig. 4 shows the performance of the GGNN-based code recommendation model with the

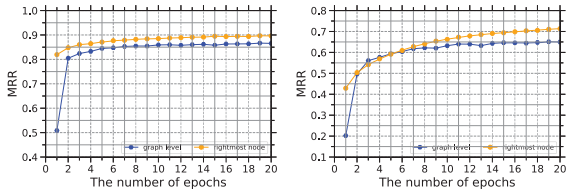
addition of two semantic edges and the preservation of only the simple structural edges of the abstract syntax tree. The experimental results show that the model with two kinds of semantic edges, data flow edge and control flow edge, shows better performance. Although the performance difference is not significant, it indicates that adding semantic information to the code model can enhance the ability of the model to understand the code.



(a) type predict (b) value predict

Figure 4: The effect of edge

Generally, graph neural network has two types of output, one for each node and the other for the whole graph. This method uses the output value of the rightmost node in the abstract syntax tree instead of the value of the whole graph when generating the probability distribution of the next node. Fig. 5 shows the performance of the model in this article using two types of output. Experimental results show that using the right node as the output in the type prediction task, MRR has reached 0.8 at the beginning of the training, similar to the performance achieved by graph-level inputs in final training. One of the important reasons is still the regularity of node types. Usually, the type of the previous node will have constraints on the type of the next node. In the node value prediction task, the gap between the output granularity of the two types is more obvious than that of the node type prediction task. Another reason why the model with the right-most node as the output not only does not lose the information of other nodes but also achieves better performance is that the multi-attention mechanism is used so that a single node can obtain the global view.



(a) type predict (b) value predict

Figure 5: Impact of output granularity

V. CONCLUSION AND FUTURE WORK

This paper analyzes the inapplicability of current smart contract development tools. In the smart contract integrated development environments, there are cases where the code recommendation function is weakly targeted and the accuracy rate is low. This paper proposes smart contract code recommendation based on graph neural network, which aims to improve the accuracy of code recommendation and help developers implement smart contracts faster and more securely.

In view of the situation that there is a large number of multiplexing of smart contracts on Ethereum and various degrees of security problems are hidden, deduplication calculations and vulnerability detection are done on the open source code of smart contracts when preprocessing data to provide high-quality data sets. In the process of program representation, in order to save the syntax and semantic information in the code as much as possible, we first use the abstract syntax tree tool to parse the source code, obtain the abstract syntax tree output, filter the abstract syntax tree output nodes according to the research problem and make node content splicing. In addition to abstracting the information of the syntax tree, this method uses the control flow information and data flow information as the edge of the node, giving the node richer characteristic information. In the design of the code model, gated graph neural networks is selected as the basic model and the multi-head attention mechanism of the Transformer is incorporated to help the model gain a global view. When making predictions for the next code word, considering that the rightmost node has the most useful information for the next node, the state of the rightmost node in the abstract syntax tree is used to map into the vocabulary to form a list of code recommendations.

Experimental results show that the model based on graph neural network proposed in this paper is better than the existing model recommended by smart contract code in terms of accuracy. Data preprocessing for smart contract features enhances model performance and quality.Strategies for building program representations help language models better understand semantics.

As there are few researches on smart contract code recommendation, the smart contract code recommendation method in this paper is a new type of work, mainly focusing on the application and basic performance of the method. Future work on smart contract code security and performance can be further studied. In addition, this paper does not cover the processing of out-of-vocabulary words, which is an in-depth branch of traditional code recommendations. At present, there are replication mechanism and open vocabulary model, which can be improved in the future.

ACKNOWLEDGMENT

The research is supported by the National Key RD Program of China (2020YFB1006002), the National Natural Science Foundation of China (62032025, 62002393). Zigui Jiang is the corresponding author.

REFERENCES

[1] Amiangshu Bosu, Anindya Iqbal, Rifat Shahriyar, and Partha Chakraborty. Understanding the motivations, challenges and needs of blockchain software developers: a survey. *Empirical Software Engineering*, 24(4):2636–2673, 2019.

[2] Simone Porru, Andrea Pinna, Michele Marchesi, and Roberto Tonelli. Blockchain-oriented software engineering: challenges and new directions. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, pages 169–171. IEEE Computer Society, 2017.

[3] Gail C Murphy, Mik Kersten, and Leah Findlater. How are java software developers using the elipse ide? *IEEE software*, 23(4):76–83, 2006.

[4] Yuan Huang, Queping Kong, Nan Jia, Xiangping Chen, and Zibin Zheng. Recommending differentiated code to support smart contract update. In *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*, pages 260–270, 2019.

[5] Manuel Araoz, Demian Brener, Francisco Giordano, Santiago Palladino, Teemu Paivinen, Alberto Gozzi, and Franco Zeoli. zeppelin_os: An open-source decentralized platform of tools and services on top of the evm to develop and manage smart contract applications securely. 2017.

[6] Meng Ren, Fuchen Ma, Zijing Yin, Ying Fu, Huizhong Li, Wanli Chang, and Yu Jiang. Making smart contract development more secure and easier. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1360–1370, 2021.

[7] Meng Ren, Fuchen Ma, Zijing Yin, Huizhong Li, Ying Fu, Ting Chen, and Yu Jiang. Scstudio: a secure and efficient integrated development environment for smart contracts. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 666–669, 2021.

[8] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. Learning to represent programs with graphs. In *International Conference on Learning Representations*, 2018.

[9] Yujia Li, Daniel Tarlow, Marc Brockschmidt, and Richard S. Zemel. Gated graph sequence neural networks. In *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*, 2016.

[10] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. volume 30, pages 5998–6008, 2017.

[11] Yanlin Wang and Hui Li. Code completion by modeling flattened abstract syntax trees as graphs. *Proceedings of AAAIConference on Artificial Intellegence*, 2021.

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

[13] Bernhard Mueller. Smashing ethereum smart contracts for fun and real profit. *HITB SECCONF Amsterdam*, 9:54, 2018.

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

[15] Kang Yang, Huiqun Yu, Guisheng Fan, Xingguang Yang, and Zijie Huang. A graph sequence neural architecture for code completion with semantic structure features. *J. Softw. Evol. Process.*, 34(1), 2022.