2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC) | 978-1-6654-1403-6/20/$31.00 ©2021 IEEE | DOI: 10.1109/ICPC52881.2021.00010

A Multi-Modal Transformer-based Code Summarization Approach for Smart Contracts

Zhen Yang[†], Jacky Keung[†], Xiao Yu[‡*], Xiaodong Gu[§], Zhengyuan Wei[†],
Xiaoxue Ma[†], and Miao Zhang[†]
[†]Department of Computer Science, City University of Hong Kong, Hong Kong, China,
{zhyang8-c, zywei4-c, xiaoxuema3-c, miazhang9-c}@my.cityu.edu.hk, Jacky.Keung@cityu.edu.hk
[‡] School of Computer Science and Technology, Wuhan University of Technology, Wuhan, China, xiaoyu@whut.edu.cn
[§]School of Software, Shanghai Jiao Tong University, Shanghai, China, xiaodong.gu@sjtu.edu.cn

***Abstract*—Code comment has been an important part of computer programs, greatly facilitating the understanding and maintenance of source code. However, high-quality code comments are often unavailable in smart contracts, the increasingly popular programs that run on the blockchain. In this paper, we propose a Multi-Modal Transformer-based (MMTrans) code summarization approach for smart contracts. Specifically, the MMTrans learns the representation of source code from the two heterogeneous modalities of the Abstract Syntax Tree (AST), i.e., Structure-based Traversal (SBT) sequences and graphs. The SBT sequence provides the global semantic information of AST, while the graph convolution focuses on the local details. The MMTrans uses two encoders to extract both global and local semantic information from the two modalities respectively, and then uses a joint decoder to generate code comments. Both the encoders and the decoder employ the multi-head attention structure of the Transformer to enhance the ability to capture the long-range dependencies between code tokens. We build a dataset with over 300K <method, comment> pairs of smart contracts, and evaluate the MMTrans on it. The experimental results demonstrate that the MMTrans outperforms the state-of-the-art baselines in terms of four evaluation metrics by a substantial margin, and can generate higher quality comments.**

***Index Terms*—Smart Contracts, Code Summarization, Transformer, Graph Convolution, Structure-based Traversal**

I. Introduction

Automatic code summarization, which generates a brief natural language description for source code, can greatly facilitate programmers in code comprehension and maintenance. Many approaches have been proposed to generate comments for some common programming languages, such as Java and Python. In recent years, smart contracts, as programs automatically running on the blockchain [1]–[3], have been applied in various business areas to enable more efficient and trustable transactions [4], [5]. More and more developers also devote themselves to the development of smart contracts, and contribute their code to various smart contract communities (e.g., Etherscan.io [6]).

However, as an increasingly universal and vital field, automatic code summarization for smart contracts has not gained much attention yet. This may cause a few vital issues: (1) We have observed that most of smart contract comments are unavailable, thus resulting in great difficulties in comprehending and learning code between developers. (2) Code clone and duplicates in smart contracts is a more common phenomenon than other softwares [4], [7]. He *et al.* [8] found that about 10% of the vulnerabilities were introduced by code clone, and the misuse of uncommented code is a principal reason. To this end, it is necessary to automatically generate high-quality code comments for smart contracts. The challenges of automatic code summariztion usually include:

(1) How to extract the semantic information of source code.

Early researchers, such as Iyer *et al.* [9] and Loyola *et al.* [10], used the plain source code as the input of the code summarization model, which ignored the structural information of source code. Therefore, most recent works firstly parsed source code to the Abstract Syntax Tree (AST), and then extracted the semantic information of code from AST. For example, Hu *et al.* [11] firstly traversed the AST by the (Structure-based Traversal) SBT method to obtain the SBT sequences, then used the plain source code and the SBT sequences as inputs to learn the semantic information of source code. LeClair *et al.* [12] regarded the AST as a graph, and used the Graph Neural Network (GNN)-based encoder to model the AST, combined with the RNN-based encoder to model the plain source code. However, ASTs can be represented as multiple modalities, such as SBT sequences and graphs, each focusing on a distinct aspect of the semantic information. Therefore, it is not comprehensive to use a single AST modality to represent the semantic information of code.

(2) How to capture the long-range dependencies between code tokens.

The source code of smart contract methods can be very long. In our experiment dataset, smart contract methods contain 58.72 code tokens on average, and the longest one contains 3,272 code tokens. Previous works often applied the Recurrent Neural Network (RNN), Long Short Term Memory (LSTM), or Gated Recurrent Unit (GRU) models to extract features from their inputs, which have shown to be difficult to capture the long-range dependencies between code tokens [13], [14].

In order to address the above challenges, in this paper, we propose a Multi-Modal Transformer-based (MMTrans) code summarization approach for smart contracts. Firstly, the

*Corresponding author.

978-1-6654-1403-6/21/$31.00 ©2021 IEEE
DOI 10.1109/ICPC52881.2021.00010

MMTrans learns the semantic information of source code from the two modalities of the AST, i.e., the SBT sequence and the graph. Specifically, the SBT sequence is globally parsed from the AST using the SBT method, which involves the global semantic information of source code. Meanwhile, MMTrans regards the AST as a graph, and employs the Graph Convolutional Neural Network (GCN) to learn representations of nodes based on their neighboring nodes, thus obtaining the local semantic information of source code. Then, the MMTrans uses a dual-encoder architecture: a SBT encoder for encoding SBT sequences to extract the **global** semantic information, and a graph encoder for encoding graphs to extract the **local** semantic information. Finally, the MMTrans uses a joint decoder to decode the outputs of the two encoders and previous generated words to produce the each time step's prediction. In addition, both the encoders and the decoder employ the multi-head attention structure of the Transformer to reinforce the capability of capturing the long-range dependencies between code tokens.

To evaluate the MMTrans, we carefully collect 347,410 <method, comment> pairs from 40,932 smart contracts on Etherscan.io [6], one of the most popular and active smart contract communities. We extensively assess the performance of the MMTrans against the three recently proposed approaches (i.e., Hybrid-DeepCom [11], code+gnn+GRU [12] , and Vanilla-Transformer [14]) on the dataset in terms of sentence-level BLEU (S-BLEU), corpus-level BLEU (C-BLEU), ROUGE-LCS F1 and METEOR. The experimental results show that the MMTrans performs better than the baselines by 17.23%-58.45% in terms of S-BLEU, by 20.74%-62.49% in terms of C-BLEU, by 6.78%-55.60% in terms of ROUGE-LCS F1, and by 10.17%-45.50% in terms of METEOR. We further conduct two groups of ablation experiments to explore the strength of MMTrans. Both the quantitative and instance analysis demonstrate that the MMTrans can generate higher-quality comments for smart contracts.

The main contributions of this paper can be summarized as follows:

- We propose a novel Multi-Modal Transformer-based (MMTrans) code summarization approach for smart contracts, which can extract both the global and local semantic information of code, and capture the long-range dependencies between code tokens to generate higher-quality comments.
- We build a dataset with totally 347,410 <method, comment> pairs for the field of smart contract code summarization. To the best of our knowledge, it is the first large-scale dataset for this task.
- We open source our replication package, including the dataset [15] and the source code [16] of the MMTrans for follow-up works.

The remainder of this paper is organized as follows. Section II presents the related work and background. Section III and Section IV elaborate on the proposed approach and data preparation. Section V and Section VI discuss the experiment design and results. Section VII demonstrates the threats to validity. Finally, Section VIII concludes the work of this paper.

II. RELATED WORK AND BACKGROUND

A. Code Summarization

Automatic code summarization approaches can be divided into two main categories, i.e., heuristic/template-driven approaches and AI/data-driven approaches [17]. Haiduc *et al.* [18], [19] for the first time coined the term "source code summarization", and proposed a heuristic-based approach, which applied text retrieval techniques to select some important keywords as the generated code comments. Following Haiduc *et al.*'s works, many researchers [19]–[26] proposed to design a set of heuristic rules or create some manually-crafted templates to generate code comments.

Due to the rapid development of deep learning technology, the recently proposed code summarization approaches are almost AI/data-driven approaches. Iyer *et al.* [9] for the first time proposed an AI/data-driven code summarization approach, which used the LSTM networks with attention to generate descriptions for C# code snippets and SQL queries. The subsequent works [10]–[12], [17], [27]–[34] almost adopted the Sequence-to-Sequence (Seq2Seq) model with attention mechanism. The major difference of the approaches are the input to the Seq2Seq model. For example, Loyola *et al.* [10] input the plain source code into the Seq2Seq model, Lu *et al.* [27] used the API sequences as the input, and Fernandes *et al.* [32] used the graph representations of source code as the input. In addition, in order to extract more information from the source code, Hu *et al*. [11], [35] and LeClair *et al*. [12], [17] proposed some multi-input Seq2Seq models. For example, Hu *et al*. [11] input the plain source code and the SBT sequences to the Seq2Seq model, in order to learn both the lexical and structural information from the source code and the AST. LeClair *et al*. [12] used the graph representation of the AST and plain source code as the inputs. However, it is not comprehensive that these works [11], [12], [29] used single AST modality to represent the semantic information of code. In addition, the proposed Seq2Seq models [9]–[12], [17], [27]–[33] mainly applied RNN, LSTM, or GRU to extract the code feature, which may fail to capture the long-range dependencies between code tokens. Therefore, Ahmad *et al*. [14] empirically investigated the advantage of using the Transformer model for the source code summarization task. However, they only adopted the plain source code as the single input, thus ignoring the structure information of source code.

B. Structure-based Traversal

Hu *et al*. [29] proposed the Structure-based Traversal (SBT) method, which converts the ASTs into specially formatted sequences by globally traversing the ASTs. Specifically, it applies the "type" and "value" of nodes to represent the structural and lexical information of code, respectively, and adopts a series of brackets to keep the AST structure to ensure the generated sequence is recoverable to the original AST. The SBT method was applied in some previous code

summarization models, such as ast-attendgru [17], Dual Model [36], and Hybrid-DeepCom [11], and was proved its powerful ability in preserving code structural and lexical information. Therefore, we regard the SBT sequence as the one modality of AST, and adopt the SBT method to represent the global semantic (i.e., both structural and lexical) information as an input of the MMTrans. Taking the smart contract snippet in the Figure 1 as an example, the method named *_tokensToSell* is firstly transformed to its AST format, and then the SBT sequence is further extracted from the AST. The non-leaf nodes are represented by their "type" (such as "FunctionDefinition" and "Block" in bold). For leaf nodes, they are represented by the format of "type_value" (such as "Visibility_private", etc.) in the original paper [29]. However, in our experiment, based on the original SBT sequences, we split the "type" and "value" for leaf nodes, and further split the camelCase and snake_case tokens of leaf nodes' "values" (such as from "_tokensToSell" to "_tokens", "To", and "Sell" in italic) to reduce the Out-of-Vocabulary (OOV) tokens, the detailed data processing methods are elaborated in Section IV-A.

C. Graph Convolutional Neural Network

The Graph Convolutional Neural Network (GCN) is designed for information propagation along the edges between nodes, where the hop, representing the layers of GCN, is a critical hyper-parameter. With the hop increasing, each node can aggregate a larger range of information from its neighbors, thereby focusing on a wider scope of local semantic information. Since the node embeddings in GCN includes both the "type" and "value" in ASTs, the GCN implies both the lexical and structural semantics of the AST integrated along the edges. Previous code summarization works, such as graph2seq [37] and code+gnn+GRU [12], also have proved the strength of GCN on locally distilling the semantic information from the AST and achieved promising results. Therefore, we regard the graph as another modality of AST, and adopt it as a parallel input of the MMTrans relative to the SBT sequences. The AST (graph) in Figure 1 is an example generated from the method named *_tokensToSell*, and shows the hop of 1,2 and 3 of the root node (i.e., "FunctionDefinition"). Intuitively, it can aggregate the information from its neighbors of "SimpleName", "Visibility", "ReturnParameters" and "Block" by the convolution of the first hop; and it can also indirectly aggregate the more extensive information from its children nodes' neighbors by increasing the hop number. More formally, the graph convolution process can be defined by the following layer-wise propagation rule:

$$H^{(l+1)} = \sigma(\tilde{A}H^{(l)}W^{(l)}), \tag{1}$$

where the H^l is the nodes embedding matrix at the layer l, the $\tilde{A} = A + I_N$ is the adjacency matrix A of a particular graph with added self-connection, I_N is the identity matrix, W^l is a layer-specific trainable weight matrix, and σ is the activation function [38].

D. Transformer-related Structures

1) Positional Encoding: Since the multi-head attention is not the recurrent structure, it needs the positional encoding to inject order information into the token embedding vectors. In this work, we follow one of the positional encoding approaches proposed by Vaswani *et al.* [13], which defines the specific pattern that model learns. This kind of positional encoding rule can be defined by the equations 2 and 3, where pos is the token position in a sequence, i is the dimension index, and d is total dimension of the token embedding vector.

$$PE_{(pos,2i)} = sin(pos/10000^{2i/d}) \tag{2}$$

$$PE_{(pos,2i+1)} = cos(pos/10000^{2i/d}) \tag{3}$$

2) Multi-head Attention: In order to pay attention from different perspectives and capture the long-range dependencies in sequences, Vaswani *et al.* [13] also introduced the multi-head attention mechanism. The details are given by the following equations:

$$q_1, ..., q_J = split(QW^Q) \tag{4}$$

$$k_1, ..., k_J = split(KW^K) \tag{5}$$

$$v_1, ..., v_J = split(VW^V) \tag{6}$$

$$head_j = Softmax(\frac{q_j k_j^T}{\sqrt{d_k}})v_j,\ j = 1, ..., J \tag{7}$$

$$MultiHead(Q, K, V) = Concat(head_1, ..., head_J)W^o \tag{8}$$

Here, the $Q \in \mathbb{R}^{Q_l \times Q_d}$, $K \in \mathbb{R}^{K_l \times K_d}$ and $V \in \mathbb{R}^{K_l \times V_d}$ represent the matrices of query, key and value, respectively, while $q_j \in \mathbb{R}^{Q_l \times q_d}$, $k_j \in \mathbb{R}^{K_l \times k_d}$, $v_j \in \mathbb{R}^{K_l \times v_d}$ represent their splitted matrices for $head_j$. Specifically, $q_d = k_d = v_d = d_{model}/J$. The $W^Q \in \mathbb{R}^{Q_d \times d_{model}}$, $W^K \in \mathbb{R}^{K_d \times d_{model}}$, $W^V \in \mathbb{R}^{V_d \times d_{model}}$ are the three trainable weight matrices. The equation 7 describes the Scaled Dot-Product Attention output of $head_j$, where the d_k is the scaling factor equals to k_d. Finally, after the concatenating from all heads and the linear transformation with $W^o \in \mathbb{R}^{Jv_d \times d_{model}}$, we obtain the output of the multi-head attention in equation 8 [13].

3) Point-Wise Feed-Forward Networks: This is another module of Transformer that applied in [13]. It is composed of two dense layers with a ReLU activation function in between, which can be defined by the equation 9, where W_1 and W_2 are the weight matrices of each layer, b_1 and b_2 are their corresponding bias, and x is the input matrix. The dimensionality of the inner-layer d_{ff} is a hyper-parameter in [13].

$$FFN(x) = max(0, xW_1 + b_1)W_2 + b_2 \tag{9}$$

III. APPROACH

The whole framework of the MMTrans illustrated in Figure 2 includes the three stages: the data processing, the MMTrans training, and the MMTrans testing. The source code we obtained from the Etherscan.io is parsed and processed into a parallel corpus of smart contract methods and their

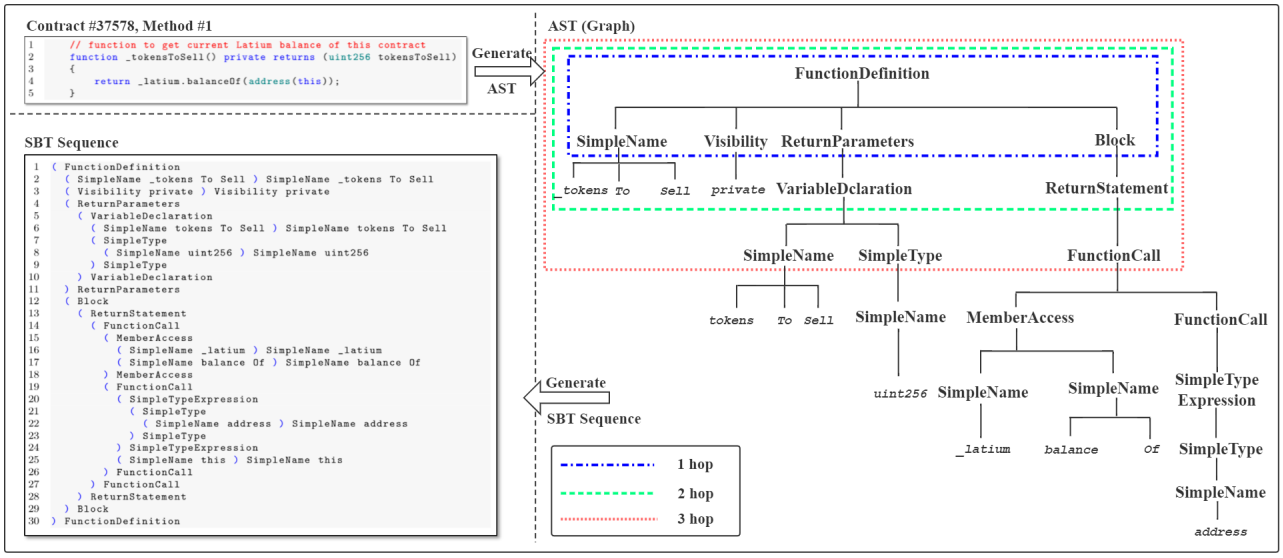

Fig. 1. The AST (Graph) and SBT Sequence of the Smart Contract Method named _tokensToSell

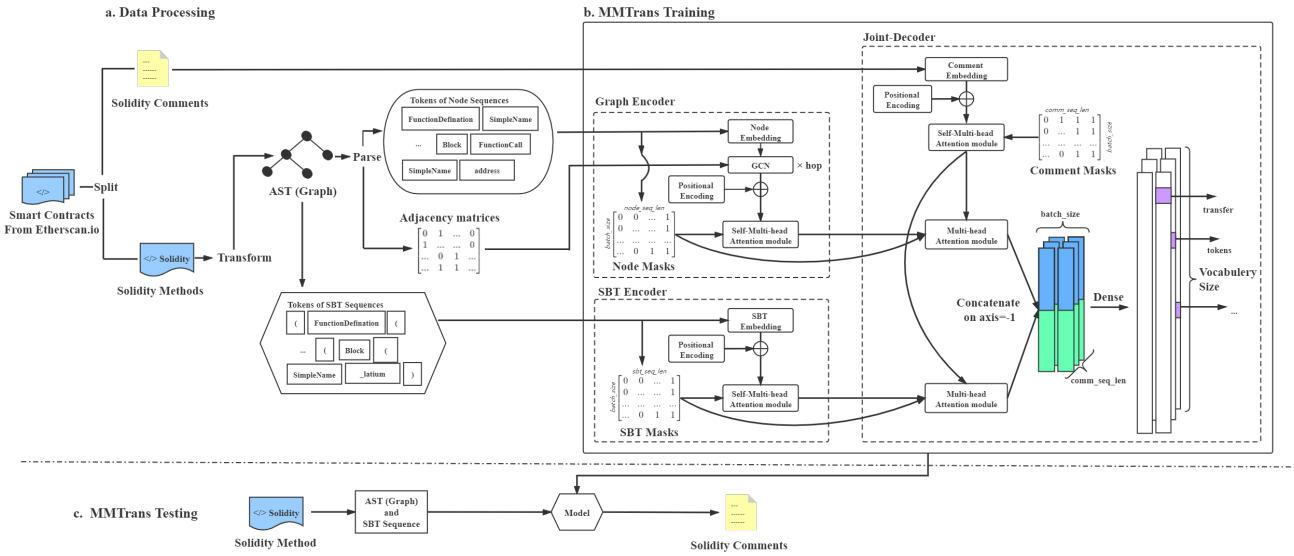

Fig. 2. The Overall Framework of MMTrans

corresponding comments. To comprehensively learn the semantic information of source code, we transform the smart contract methods to graphs (i.e., ASTs) and SBT sequences, respectively, as the inputs of the MMTrans. The MMTrans consists of the two encoders (i.e., the graph encoder and SBT encoder) and a joint decoder. The node sequences and edges (i.e., their corresponding adjacency matrices) of graphs are fed into the graph encoder to learn the local semantic information, while the SBT sequences are fed into the SBT encoder to learn the global semantic information. Subsequently, in the training stage, the joint decoder integrates comment sequences, the graph encoder outputs, and the SBT encoder outputs to produce a batch of sentences under the teacher forcing method. Finally, the back-propagation is executed based on the predefined loss function to optimize the whole network. However, in the testing stage, the joint decoder integrates the previously generated comment words, and above two encoder outputs to predict one word at each time step. It is noticeable that we do not include plain source code as one part of inputs for the MMTrans to learn the lexical information of source code, because each of the modality of AST we adopt already contains both the lexical and structural information, as

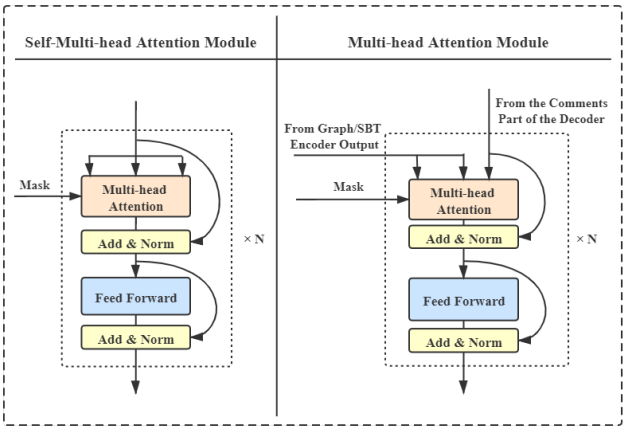

Fig. 3. (Self) Multi-head Attention Module

mentioned in II-B and II-C.

A. Graph Encoder

Initially, for a batch of node sequences $X \in \mathbb{R}^{N_{batch} \times l}$, where l represents the maximum length of node sequences of this batch and N_{batch} represents the batch size, the graph encoder firstly embeds the node sequences X with the embedding size $d = 256$. Then, the GCN layer takes the embedding layer output and the edges $E \in \mathbb{R}^{N_{batch} \times l \times l}$ as the inputs to perform the graph convolution that we described in the Section II-C. Previous work by LeClair *et al.* [12] has proved that the graph convolution layers $hop = 2$ is the best setting for the code summarization task, here we follow their setup and fix the $hop = 2$. Each node at the end of the GCN layer aggregates the neighboring information after the graph convolution. Since we adopt the pre-order traversal method to produce the node sequences from ASTs, the node sequences imply the original appearance order of the tokens in the source code, which is also the necessary information that needs to be considered. Therefore, we add the positional encoding matrices $PE \in \mathbb{R}^{N_{batch} \times l \times d}$ to the output of the GCN layer, so as to inject the position-wise information.

Subsequently, the output of the GCN layer is imported into the Self-Multi-head Attention Module (SMAM) to distill their semantic information further. The internal structure of the SMAM is demonstrated in Figure 3. It includes the multi-head attention, layer-normalization, and point-wise feed-forward network (the d_{ff} of the network is set to 512), which are the principal parts of the Transformer introduced in Section II-D. Initially, we set the head number $J = 4$, so that the multi-head attention can focus on each node sequence from four different representation subspaces [13]. Moreover, the d_{model} in the multi-head attention is set to 256, representing the width of this module; while the number of layers N, representing the depth of this module, is set to 1. Besides, we also adopt the node mask $M \in \mathbb{R}^{N_{batch} \times l}$ generated from the batch data to avoid distracting attention by <PAD> tokens. Finally, we obtain the output of SMAM $\hat{X} \in \mathbb{R}^{N_{batch} \times l \times d_{model}}$ at the end of the graph encoder, and the whole process can be described by the following equation, where the f is the abstract mapping function constructed by the graph encoder:

$$\hat{X} = f(X,\ E,\ PE,\ M) \tag{10}$$

B. SBT Encoder

For a batch of SBT sequences $X^{'} \in \mathbb{R}^{N_{batch} \times l^{'}}$, where the $l^{'}$ represents the maximum length of SBT sequences of this batch, the SBT encoder also firstly embeds the $X^{'}$ with the embedding size $d = 256$ and injects the position-wise information with $PE^{'} \in \mathbb{R}^{N_{batch} \times l^{'} \times d}$. Subsequently, the SBT encoder adopts the SMAM to extract the semantic information with the same hyper-parameters and uses the SBT mask $M^{'} \in \mathbb{R}^{N_{batch} \times l^{'}}$ to avoid distraction. Thereby, we obtain the final output of SBT encoder $\hat{X}^{'} \in \mathbb{R}^{N_{batch} \times l^{'} \times d_{model}}$. The equation below illustrates the abstract mapping function $f^{'}$ of the SBT encoder:

$$\hat{X}^{'} = f^{'}(X^{'},\ PE^{'},\ M^{'}) \tag{11}$$

C. Joint Decoder

Similarly, for a batch of comment sequences $Y \in \mathbb{R}^{N_{batch} \times l^{Y}}$, where the l^{Y} represents the maximum length of comment sequences of this batch, the joint decoder also firstly embeds the Y with the embedding size $d = 256$, and injects the positional information $PE^{Y} \in \mathbb{R}^{N_{batch} \times l^{Y} \times d}$. Then, the features of comment sequences are extracted by the SMAM with the same hyper-parameters. It should be noticed that the comment mask $M^{Y} \in \mathbb{R}^{N_{batch} \times l^{Y} \times l^{Y}}$ is an addition of a padding mask and a look-ahead mask (i.e., an upper triangular matrix), which is used for avoiding distraction and information leakage of the subsequent tokens in training [13].

Next, two Multi-head Attention Modules (MAMs) are introduced in the joint decoder with the same hyper-parameters to SMAM. The internal structure is shown in Figure 3. One for comment sequences and the output of graph encoder $\hat{X}$, another for the comment sequences and the output of SBT encoder $\hat{X}^{'}$, thereby learning which tokens from the two encoders are important to the inference of comments, respectively. The outputs from the two MAMs are concatenated together on their last axis, representing a merge of their respective prediction for the comments. Finally, the merged output is fed into a linear transformation layer followed by a softmax function to produce the probability distribution over the vocabulary, thereby obtaining the final result $\hat{Y} \in \mathbb{R}^{N_{batch} \times l^{Y} \times S}$ of the MMTrans, where the S represents the comment vocabulary size. The outlined mapping function f^{Y} of joint decoder can be summarized by the equation below:

$$\hat{Y} = f^{Y}(Y,\ \hat{X},\ \hat{X}^{'},\ PE^{Y},\ M^{Y}) \tag{12}$$

Similar to most of the Seq2Seq models, we define the loss function for each batch as equation 13, where the l_i^y represents the length of the ith real comment (ground truth) removed <PAD> tags, and $p(\hat{Y}_{ij}^{(z)})$ represents the probability that the

jth token in the ith sample is the zth (z is the ground truth) word in the whole comment vocabulary.

$$Loss_{batch} = -\frac{1}{N_{batch}} \sum_{i=1}^{N_{batch}} \frac{1}{l_i^y} \sum_{j=1}^{l_i^y} log\ p(\hat{Y}_{ij}^{(z)}) \qquad (13)$$

IV. Data Preparation

A. Preprocessing

The raw dataset is provided by Zhuang *et al.* [39], which was collected from the Etherscan.io. The dataset contains totally 40,932 Ethereum smart contracts written in solidity with 933,146 *normal methods*, 73,533 *modifiers*, and 12,482 *fallback methods*. *Normal methods* includes *constructors* and other *functional methods*; *Modifiers* are used to change the behaviour of functions in a declarative way, such as checking a condition prior to executing the function; *Fallback methods* will be executed on a call to the contract, if none of the other methods match the given method signature; another kind of method in solidity is the *receive method*, which is also a kind of *fallback method*, and is first introduced in the version of solidity 0.6.0 to receive ether [40]. According to our preliminary study on the whole dataset, we find 1-to-*n* matching problem between methods and comments, i.e., the same code may correspond to different comments among most of the *fallback methods*, which will confuse the MMTrans in training. The main reason is that most of the developers use *fallback methods* for reverting. But different *fallback methods* in the different smart contracts will revert different objectives. In addition, generating comments for *constructors* and *receive methods* is trivial, because their functionalities are fixed and easy for machines to learn. Furthermore, there is also no *receive method* in the whole dataset. Therefore, in our experiment, we only consider normal *functional methods* and *modifiers*, and we remove the methods without comments. The remaining data are formulated as <method, comment> pairs for further process.

According to the introduction of annotation in the solidity doc [40] and our observation, we find that smart contract developers tend to place their comments under the NatSpec tags by the priority order of $@notice$, $@dev$, $@return$ or just "//" and "/**/". Therefore, we extract the texts behind the $@notice$ tag firstly, if there is no $@notice$ tag, we extract the texts behind the $@dev$ tag, and so on. In addition, following the prior similar work [11], [12], [29], [36], we also use the first sentence of texts as the ground truth comment, which typically describes the functionality of the particular method. Moreover, we remove those <method, comment> pairs whose comments contain less than 4 words for better computation of the BLEU-4 score [11]. Finally, we collect the 347,410 <method, comment> pairs from the 40,932 smart contracts.

B. Data Transformation

We transform the source code to SBT sequences and graphs (represented by xml format) respectively by utilizing the solidity-parser-antlr [41]. To reduce the Out-Of-Vocabulary (OOV) tokens and facilitate the model to capture token representation, we split the camelCase and snake_case tokens for the "value" of leaf nodes in both SBT sequences and graphs. Further, we parse the graphs to their corresponding node sequences and edges (i.e., adjacency matrices), and formulate the <method, comment> pairs to (SBT sequence, node sequence, adjacency matrix, comment) tuples. The statistics of the lengths of the SBT sequences, graph node sequences, and comments are shown in the Table I. The average length of the SBT sequences, graph node sequences and comments are 241.44, 75.92, and 10.04, respectively. Here, we set the maximum length of comments to 20, and remove those tuples with comment length larger than 20 (i.e., retain 96.53% of the whole data). We also set the maximum length of SBT sequences and graph nodes to 600 and 200, and truncate those excessively long sequences. Afterward, we replace the $numeral$ and $string$ in source code with <NUM> and <STR> respectively. Since the $address$ of smart contracts is a special object in smart contracts and is composed of fixed 40 hexadecimal digits, we generalize $address$ constants by <ADDR>. Finally, for the SBT sequences and comments, we add <START> and <END> tokens to represent the start and end of sequences. However, for graph nodes sequences, we keep them intact, which is also a common operation in GCN [12].

TABLE I
STATISTICS FOR DATA LENGTHS

The Lengths of the SBT Sequences					
Avg.	Mode	Median	≤ 200	≤ 400	≤ 600
241.44	69	163	59.49%	85.69%	94.92%
The Lengths of the Graph Node Sequences					
Avg.	Mode	Median	≤ 100	≤ 150	≤ 200
75.92	21	52	76.77%	89.98%	95.27%
The Lengths of the Comments					
Avg.	Mode	Median	≤ 10	≤ 20	≤ 30
10.04	8	10	57.18%	96.53%	99.44%

C. Generating Vocabularies and Input Pipeline

The above procedures yield 317,680 tuples. We randomly select 90% (285,912 samples) of them for training, 5% (15,884 samples) of them for validation, and 5% (15,884 samples) of them for testing. We remove duplicated samples in the validation set and the testing set that are already included in the training set to avoid data leakage, and finally remain 1,185 and 1,159 samples, respectively. Then, we generate vocabularies for the SBT sequences, the graph node sequences and the comments, respectively on the training set. The size of above vocabularies are 10441, 10431, and 13174, respectively. Furthermore, we set a <UNK> token in each of the above vocabulary to substitute the OOV tokens in the validation set and the testing set.

To prepare the data input pipeline, we randomly select 100 samples to form each batch without replacement. For

each input tunnel (i.e., SBT sequences, graph node sequences, adjacency matrices, and comment sequences) of each batch, we append the special tag <PAD> to pad them to the maximum length of their batch.

V. EXPERIMENT DESIGN

In this section, we propose our research questions and the corresponding methodologies. Meanwhile, we elaborate on the baselines from the recent similar works, the evaluation metrics, and the experimental devices.

A. Research Questions

Our evaluation aim to verify if the MMTrans adopting the multi-modalities of AST and the multi-head attention structure outperforms the state-of-the-art baselines, and why the above two points cause the outperformance of the MMTrans. Based on these concerns, we propose the following Research Questions (RQs):

- RQ_1**:** How effective is the MMTrans compared with the state-of-the-art baselines introduced in Section V-C?
- RQ_2**:** How does the head number of the multi-head attention structure affect the performance of the MMTrans?
- RQ_3**:** What are the advantages of using the multi-modalities of AST and the multi-head attention structure?

Inspired by the previous works [11], [12], [29], we make the first attempt of utilizing multi-modalities of AST (i.e., both SBT sequences and graphs) and the multi-head attention structure to construct the framework of dual encoders along with a joint decoder, thereby proposing the MMTrans. To this end, the RQ_1 is put forward to evaluate the MMTrans against other state-of-the-art models in terms of the metrics in Section V-D.

On the other hand, the head number in the multi-head attention structure has shown to be important in attention allocation from different representation subspaces. To this end, we put forward the RQ_2 in this work to explore how the head number affects the MMTrans learning in smart contracts code summarization.

Finally, the aim of the RQ_3 is to investigate why the MMTrans is more effective in generating code comments, and how its heterogeneous multi-modalities of AST and the multi-head attention structure contribute to the generation of higher-quality comments for smart contracts, respectively.

B. Methodology

To answer the RQ_1, we reproduced three baselines that highly related to our work. Following LeClair *et al.*'s [12] experiment setup, we try to set the hyper-parameters of the baselines consistent with the MMTrans for the fair comparison. The detailed hyper-parameters setting for the MMTrans has been elaborated in Section III. Besides, we adopt the same training strategy for each model during the experiment. Specifically, we set the maximum training epoch to 50. Each model is validated every 500 minibatches on the validation set by sentence-level BLEU. We save the model with the best validation performance, and adopt early stopping with the patience of 5 to avoid the model overfitting and save computation cost. Furthermore, we adopt Adam [42] as the training optimizer, and follow the learning rate decay schedule in [13]. After the above training operations for each model, we choose their respective best performing model to evaluate on the testing set in terms of the metrics in Section V-D, and report the comparison result in this paper.

For the RQ_2, we fix the other hyper-parameters of the MMTrans, and set the head number $J = \{2, 4, 8, 16, 32\}$ respectively. For each adjustment of J, we retrain the MMTrans following the training strategy and report the experimental result in terms of the automated metrics in this paper. The reason for the choice of above head numbers is that they should be divisible by d_{model}, which is a necessary prerequisite for the multi-head attention structure.

Finally, for the RQ_3, we evaluate the strength of the heterogeneous multi-modalities of AST and the multi-head attention structure by ablation experiments separately. (1) Firstly, we construct an incomplete-MMTrans (i-MMTrans) utilizing the graph and plain source code as double inputs. Then, we adopt the i-MMTrans to compare with the third baseline (i.e., vanilla-Transformer with only the plain source code input) and the MMTrans with both SBT sequences and graphs as inputs, thereby exploring the contribution of SBT sequences and graphs, respectively. (2) However, for the evaluation of the strength of the multi-head attention structure, we adopt the comparison between the i-MMTrans and the second baseline, i.e., code+gnn+GRU, because they both adopt the graph and plain source code as double inputs. But the i-MMTrans adopts the multi-head attention structure, while the code+gnn+GRU adopts the GRU-based structure.

C. Baselines

We compare the MMTrans with the three baselines that are published in last year and directly related to our work. We list their detailed information as below:

(1) Hybrid-DeepCom: Hu *et al.* [11] exploited the plain source code and SBT sequence with only nodes' "type" as the double inputs to extract structural and lexical information of source code, respectively. They also construct a GRU-based Seq2Seq model to process the double inputs and generate comments of Java methods. This approach was published in the EMSE volume 25, 2020, and is the latest representative approach that adopts SBT sequences as input separately with the plain source code.

(2) code+gnn+GRU: LeClair *et al.* [17] proposed to utilize the ASTs (graphs) and the plain source code as the double inputs to improve the code summarization performance based on their previous work [17] published in the ICSE'2019. This approach was presented in the ICPC'2020 and is the latest work adopting GCN in the code summarization task.

(3) Vanilla-Transformer: Ahmad *et al.* [14] proposed to adopt the Transformer to solve the Java source code comments generation task, which is the first attempt of utilizing the Transformer in this field. This work was published in

the ACL'2020. We adopt their base model (i.e., Vanilla-Transformer) to involve in the comparison, rather than their full model with the relative positional encoding and copy attention, because these tricks make it difficult to distinguish the improvement of multi-modalities of AST on transformer-based approach.

Since those baselines are all applied to Java methods, we change their inputs as our smart contracts data, and follow their data processing steps to prepare their inputs. Besides, we adopt the greedy search algorithm in inference for all approaches to save the computation cost and ensure a fair comparison.

Other noteworthy recent works, such as Wang *et al*. [43], Zhang *et al*. [34], adopted the Reinforcement Learning (RL) and Information Retrieval (IR) techniques based on recurrent models to improve the model performance, respectively. Since the second improvement of the MMTrans focuses on the structural upgrade by the Transformer towards the recurrent models, the contribution of the Transformer-based structure towards the recurrent models cannot be distinguished from the RL and IR techniques based on recurrent models. Therefore, the above two works are not suitable for comparison in this experiment. However, a potential future direction is to study the effect of the RL and IR techniques combining with the Transformer-based structure in another separate experiment.

D. Metrics

We adopt BiLingual Evaluation Understudy (BLEU) [1] [44], Recall-Oriented Understudy for Gisting Evaluation (ROUGE) [2] [45], and Metric for Evaluation of Translation with Explicit ORdering (METEOR) [1] [46] to evaluate the performance of the code summarization approaches.

- We report a composite BLEU score, which is the average of BLEU-1, BLEU-2, BLEU-3, and BLEU-4 (BLEU-n is the n-gram precision of a candidate sequence to the reference). Following Hu *et al*.'s work [11], we adopt both sentence-level BLEU (S-BLEU) and corpus-level BLEU (C-BLEU) to evaluate the generated comments, respectively. The S-BLEU calculates the composite BLEU score according to the sentence, and the C-BLEU calculates the composite BLEU score according to the whole corpus. In addition, to avoid non-overlapping n-grams in sentences, we simply use the smoothing-1 method [47] to assist the computation of S-BLEU.
- ROUGE evaluates how much of reference text appears in the generated text, which can be thought of as a recall score. Following LeClair *et al*.'s work [12], we adopt the ROUGE-LCS (Longest Common Sub-sequence) F1 to evaluate the generated comments on the LCS matching degree between generated comments and references.
- METEOR is also a widely used recall-oriented metric in machine translation and code summarization tasks. It evaluates how well the generated comments capture content from the references via recall, which is computed by stemming and synonymy matching.

[1]NLTK: (https://www.nltk.org/) is used to calculate BLEU and METEOR
[2]rouge: (https://github.com/pltrdy/rouge) is used to calculate ROUGE

E. Experimental Device

The experiments are conducted on a Ubuntu GPU server with four RTX2080ti GPUs of 11 GB memory for each. Our proposed MMTrans is constructed by Tensorflow 2.3 based on CUDA 10.1 and cuDNN 7.4.

VI. RESULTS

This section reports the experimental results for the research questions proposed in Section V-A.

A. RQ_1: Quantitative Evaluation

Table II shows the performance of the MMTrans ($J = 4$) and the compared baselines. As shown in the table, the MMTrans performs the best, and achieves a S-BLEU score of 30.47, a C-BLEU score of 34.14, a ROUGE-LCS F1 score of 50.57, and a METEOR score of 43.24. Specifically, the MMTrans outperforms the Vanilla-Transformer by 17.23% in terms of S-BLEU, by 20.74% in terms of C-BLEU, by 6.78% in terms of ROUGE-LCS F1, and by 10.17% in terms of METEOR; the MMTrans also outperforms the code+gnn+GRU by 38.94%, 39.23%, 55.60%, and 21.22% in terms of the four metrics, respectively; and the MMTrans outperforms the Hybrid-DeepCom by 58.45%, 62.49%, 31.18%, and 45.05% in terms of the four metrics, respectively. Towards the great improvement of the MMTrans against the three baselines, we generally attribute it to the multi-head attention structure and the utilization of both GCN and SBT to capture the local and global semantic information of code. We will discuss the strength of the MMTrans in more depth in Section VI-C. We also find that the code+gnn+GRU outperforms the Hybrid-DeepCom in terms of S-BLEU, C-BLEU, and METEOR, while the latter performs better in terms of ROUGE-LCS F1. This may owe to the higher attention of GCN on specific tokens and structures [12], leading to the higher score in terms of the gram-statistic-oriented metrics (i.e., S-BLEU, C-BLEU, and METEOR). However, SBT sequences represent the global semantic information of source code, therefore, the Hybrid-DeepCom can capture the global semantic information, and performs better in terms of the longest common sub-sequence matching (measured by ROUGE-LCS F1). Moreover, even the Vanilla-Transformer only adopts the plain source code as the single input, its performance is still better than that of the Hybrid-DeepCom and code+gnn+GRU by a relatively large margin, which indicates the powerful capability of the Transformer model.

Answer: The MMTrans outperforms the Hybrid-DeepCom, code+gnn+GRU and Vanilla-Transformer by a significant margin in terms of all the automated evaluation metrics.

B. RQ_2: Head Number Analysis

As mentioned in Section V-A, the head number J is the prominent part in the multi-head attention module compared with other attention structures, and affects the model performance to a great extent. In order to explore the influence of the J in the MMTrans, we fix the other hyper-parameters and tune the $J = \{2, 4, 8, 16, 32\}$, respectively. The automated

TABLE II
AUTOMATED METRICS EVALUATION RESULTS FOR THE BASELINES AND MMTRANS

Baseline	head number (J)	S-BLEU*(%)	C-BLEU*(%)	ROUGE-LCS F1 (%)	METEOR (%)
Hybrid-DeepCom	/	19.23	21.01	38.55	29.81
code+gnn+GRU	/	21.93	24.52	32.50	35.67
Vanilla-Transformer	4	25.99	28.79	47.36	39.25
Ours					
i-MMTrans	4	28.67	30.77	48.76	41.51
MMTrans	2	23.57	26.63	43.45	35.37
MMTrans	4	**30.47**	**34.14**	**50.57**	**43.24**
MMTrans	8	29.68	33.19	50.50	43.26
MMTrans	16	22.27	24.91	42.71	34.38
MMTrans	32	26.97	30.41	47.38	39.67

* S-BLEU represents the Sentence-level BLEU score; C-BLEU represents the Corpus-level BLEU score.

metric evaluation results are presented in Table II. The whole experiment result presents a general trend of increasing first then decreasing in terms of each of the evaluation metrics. And the performance reaches the peak when the $J = 4$. With the J increasing, more representation subspaces of tokens appear but the embedding dimension of tokens decreases. As such, a potential explanation is that increasing the head number indeed can focus on more different perspectives; but when the subspaces proliferate excessively, the low representation dimension makes it impossible to accurately describe tokens, thus leading the attention distraction. Also notice that using head number $J = 32$ outperforms the $J = 16$, this may due to the random initialization or other minor factors.

Answer: When tuning the head number $J = \{2, 4, 8, 16, 32\}$, there is a general performance trend of increasing first then decreasing, and the MMTrans performs the best when the head number is 4.

C. RQ_3: Strength of the MMTrans

This section elaborates on the strength of MMTrans from two perspectives, i.e., heterogeneous multi-modalities of AST and the multi-head attention structure.

(1) The Vanilla-Transformer uses the plain source code as a single input; the i-MMTrans has double inputs of graphs and plain source code, while the MMTrans has double inputs of graphs and SBT sequences. Moreover, these approaches all employ the multi-head attention structure. As shown in Table II, we find that the i-MMTrans outperforms the Vanilla-Transformer by 10.31% in terms of S-BLEU, by 6.88% in terms of C-BLEU, by 2.96% in terms of ROUGE-LCS F1, and by 5.76% in terms of METEOR, which indicates that the additional input of graphs indeed boosts the model performance. Subsequently, we make a comparison between the i-MMTrans and the MMTrans. The Table II demonstrates that the latter outperforms the former by 6.28%, 10.95%, 3.71%, and 4.17% in terms of the four metrics respectively, which proves that the additional input by SBT sequences can further improve the model performance.

Meanwhile, we present the first two instances to intuitively show the strength of the heterogeneous multi-modalities of AST in Table III. Following LeClair *et al.*'s work [12], we intuitively illustrate the power of multi-modalities of AST from the copy mechanism perspective. For the first instance, the i-MMTrans can directly copy the correct words, such as "modifier", "when", "crowdsale" and "ended", from the source code to the comment. Comparing with the Vanilla-Transformer, its copy capability is indeed more powerful. However, the i-MMTrans seems relatively weaker in summarizing the main idea of source code, when the incorrect words accounts for a relatively large ratio. Noticing the second instance, the i-MMTrans wrongly copies the word "sell" and "tokens", which appear with the highest frequency in the source code, therefore does not capture the main idea of the code snippet. A potential explanation is the i-MMTrans equipped with GCN puts more focus on local features of the AST, causing it easy to be attracted by the specific words and code structure. However, the MMTrans equipped with both the GCN and SBT can weigh between local and global semantic information, therefore modifies the fault caused by the i-MMTrans, and catches the correct main idea of the source code in the second instance.

(2) As mentioned in Section V-B, the i-MMTrans and code+gnn+GRU both exploit the graph and plain source code as the double inputs, but the former adopts the multi-head attention structure while the latter adopts the GRU-based structure. Statistically, the i-MMTrans outperforms the code+gnn+GRU by 30.73% in terms of S-BLEU, by 25.49% in terms of C-BLEU, by 50.03% in terms of ROUGE-LCS F1, and by 16.37% in terms of METEOR, which indicates that applying the multi-head attention structure indeed improves the performance a lot in the source code summarization task. And it is noticeable that the contribution to the performance improvement of the multi-head attention structure is greater than that of the multi-modalities of AST. We also list two examples of the generated comments in the Table III. The instance #3 is a short method that can be parsed to the plain source code with 49 tokens and the graph node sequence with 59 tokens. The i-MMTrans and code+gnn+GRU can both generate the exact correct comments on this short sample. Nevertheless, when it comes to a relatively long method, such as the instance #4 with 152 tokens in the plain source code

TABLE III
EXAMPLES OF GENERATED COMMENTS BY EACH APPROACH

ID	Smart Contract Methods	Comments

1	// Contract #32527, Method #7
	modifier whenCrowdsaleNotEnded {
	require(deadline >= now);
	_;
	}
		Vanilla-Transformer: modifier to make a function callable only when the contract is not finalized.
		i-MMTrans: modifier to make a function callable only when the crowdsale has not ended.
		MMTrans: modifier to allow actions only when the crowdsale is not ended.
		Reference: modifier to allow actions only when the crowdsale has not ended.

2	// Contract #37578, Method #1
	function _tokensToSell() private returns (uint256 tokensToSell) {
	return latium.balanceOf(address(this));
	}
		Vanilla-Transformer: function to sell tokens.
		i-MMTrans: function to sell tokens (with decimals) that we are selling tokens.
		MMTrans: function to get amount of Latium tokens (with decimals) of this contract.
		Reference: function to get current Latium balance of this contract.

3	// Contract #6743, Method #27
	function finalize() public inState(State.Success)
	onlyOwner stopInEmergency {
	if(finalized) throw;
	if(address(finalizeAgent) != 0) finalizeAgent.finalizeCrowdsale();
	finalized = true;
	}
		code+gnn+GRU: finalize a succcesful crowdsale.
		i-MMTrans: finalize a succcesful crowdsale.
		Reference: finalize a succcesful crowdsale.

4	// Contract #16118, Method #5
	function playerMakeBet(uint minRollLimit, uint maxRollLimit,
	bytes32 diceRollHash, uint8 v, bytes32 r, bytes32 s) public payable
	gameIsActive betIsValid(msg.value, minRollLimit, maxRollLimit) {
	if (playerBetDiceRollHash[diceRollHash] != 0x0
	|| diceRollHash == 0x0) throw;
	tempBetHash = sha256(diceRollHash, byte(minRollLimit),
	byte(maxRollLimit), msg.sender);
	if (casino != ecrecover(tempBetHash, v, r, s)) throw;
	. . .
	playerProfit[diceRollHash] = getProfit(msg.value, tempFullprofit);
	if (playerProfit[diceRollHash] > maxProfit) throw;
	. . .
	LogBet(diceRollHash, playerAddress[diceRollHash],
	playerProfit[diceRollHash], playerToJackpot[diceRollHash],
	playerBetValue[diceRollHash], playerMinRollLimit[diceRollHash],
	playerMaxRollLimit[diceRollHash]);
	}
		code+gnn+GRU: appends the bid's.
		i-MMTrans: public function player submit bet only if game is active bet is valid can be called.
		Reference: public function player submit bet only if game is active bet is valid.

and 200 tokens in the graph node sequence (both have been truncated to the maximum length of the above two sequences), the code+gnn+GRU with the GRU-based structure cannot summarize the correct comment, while the i-MMTrans can still generate readable and meaningful comment. The reason is that the multi-head attention structure can properly allocate different attention weights on the tokens at each time step, and summarize their key information for inference; however, the GRU-based network has a limited capability in capturing the long-range dependencies between code tokens, thus generating the low-quality comments for relatively long methods.

Answer: Leveraging SBT sequences and graphs to extract the global and local semantic information of code, and employing the multi-head attention structure to capture the long-range dependencies between code tokens contribute to the generation of higher-quality comments.

VII. THREATS TO VALIDITY

We have identified the following threats to validity:

Dataset quality: As the first smart contract code summarization work, we used some heuristic rules to extract the <method, comment> pairs according to the characteristic of smart contract data. Although we did a rigorous data processing, there may be still some noise. We will continuously refine and update the version of the open-source dataset.

Comparison on smart contract dataset: Since our work focuses on the smart contract code summarization, we did not compare the MMTrans with Hybrid-DeepCom, Vanilla-Transformer, and code+gnn+GRU on their datasets. But the results on smart contracts also have proved the effectiveness of the MMTrans. In the future, we will extend our experiment on other programming languages (e.g., Java and Python).

Fair comparison threat: Due to the hardware limitations, we were unable to conduct fully extensive hyper-parameters optimization for all baselines. Following LeClair *et al.*'s work [12], we try to mitigate the impact of this issue by making each baseline's hyper-parameters consistent with our model.

Automated evaluation: We adopt four automated evaluation metrics that have been widely used in previous code summarization studies [11], [12], [14], [29]. Although the metrics are not representative of human judgment [48], they can evaluate the performance of code summarization models quickly and quantitatively. In the future, we will conduct human evaluation on the models.

VIII. CONCLUSION

This work aims to help programmers comprehend the meaning of smart contract code by automatically generating high-quality comment. To tackle this task, we for the first time collect a smart contract code summarization dataset with 347,410 <method, comment> pairs. Meanwhile, we propose a code summarization approach named MMTrans,

which leverages the two modalities of the AST (i.e., SBT sequences and graphs) to represent both global and local semantic information of source code, then employs the two encoders and a joint decoder with the multi-head attention structure to capture the long-range dependencies between code tokens. The comprehensive experiments on the collected dataset show that the MMTrans performs better than the state-of-the-art baselines by a significant margin, and can generate higher-quality comments in the practical tests.

ACKNOWLEDGMENT

This work is supported in part by the General Research Fund of the Research Grants Council of Hong Kong (No. 11208017) and the research funds of City University of Hong Kong (7005028 and7005217), and the Research Support Fund by Intel (9220097), and funding supports from other industry partners (9678149,9440227, 9229029, 9440180 and 9220103).

REFERENCES

[1] M. Röscheisen, M. Baldonado, K. Chang, L. Gravano, S. Ketchpel, and A. Paepcke, "The stanford infobus and its service layers: Augmenting the internet with higher-level information management protocols," in *Digital Libraries in Computer Science: The MeDoc Approach*. Springer, 1998, pp. 213–230.

[2] D. Tapscott and A. Tapscott, *Blockchain revolution: how the technology behind bitcoin is changing money, business, and the world*. Penguin, 2016.

[3] A. Savelyev, "Contract law 2.0:'smart'contracts as the beginning of the end of classic contract law," *Information & Communications Technology Law*, vol. 26, no. 2, pp. 116–134, 2017.

[4] Z. Gao, L. Jiang, X. Xia, D. Lo, and J. Grundy, "Checking smart contracts with structural code embedding," *IEEE Transactions on Software Engineering*, 2020.

[5] T. Sun and W. Yu, "A formal verification framework for security issues of blockchain smart contracts," *Electronics*, vol. 9, no. 2, p. 255, 2020.

[6] "Ethereum (eth) blockchain explorer," https://etherscan.io/, 01 2021, (Accessed on 01/26/2021).

[7] Z. Yang, J. Keung, M. Zhang, Y. Xiao, Y. Huang, and T. Hui, "Smart contracts vulnerability auditing with multi-semantics," in *2020 IEEE 44th Annual Computers, Software, and Applications Conference (COMPSAC)*. IEEE, 2020, pp. 892–901.

[8] N. He, L. Wu, H. Wang, Y. Guo, and X. Jiang, "Characterizing code clones in the ethereum smart contract ecosystem," in *International Conference on Financial Cryptography and Data Security*. Springer, 2020, pp. 654–675.

[9] S. Iyer, I. Konstas, A. Cheung, and L. Zettlemoyer, "Summarizing source code using a neural attention model," in *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 2016, pp. 2073–2083.

[10] P. Loyola, E. Marrese-Taylor, and Y. Matsuo, "A neural architecture for generating natural language descriptions from source code changes," *arXiv preprint arXiv:1704.04856*, 2017.

[11] X. Hu, G. Li, X. Xia, D. Lo, and Z. Jin, "Deep code comment generation with hybrid lexical and syntactical information," *Empirical Software Engineering*, vol. 25, no. 3, pp. 2179–2217, 2020.

[12] A. LeClair, S. Haque, L. Wu, and C. McMillan, "Improved code summarization via a graph neural network," in *Proceedings of the 28th International Conference on Program Comprehension*, ser. ICPC '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 184–195. [Online]. Available: https://doi.org/10.1145/3387904.3389268

[13] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in neural information processing systems*, 2017, pp. 5998–6008.

[14] W. Ahmad, S. Chakraborty, B. Ray, and K.-W. Chang, "A transformer-based approach for source code summarization," in *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*. Online: Association for Computational Linguistics, Jul. 2020, pp. 4998–5007. [Online]. Available: https://www.aclweb.org/anthology/2020.acl-main.449

[15] "Smart contract code summarization dataset — zenodo," https://zenodo.org/record/4587089#.YEog9-gzYuV, (Accessed on 03/11/2021).

[16] "yz1019117968/icpc-21-mmtrans: Mmtrans for smart contract code summarization," https://github.com/yz1019117968/ICPC-21-MMTrans, (Accessed on 03/11/2021).

[17] A. LeClair, S. Jiang, and C. McMillan, "A neural model for generating natural language summaries of program subroutines," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 795–806.

[18] S. Haiduc, J. Aponte, L. Moreno, and A. Marcus, "On the use of automated text summarization techniques for summarizing source code," 2010.

[19] S. Haiduc, J. Aponte, and A. Marcus, "Supporting program comprehension with source code summarization," in *2010 acm/ieee 32nd international conference on software engineering*, vol. 2. IEEE, 2010, pp. 223–226.

[20] B. P. Eddy, J. A. Robinson, N. A. Kraft, and J. C. Carver, "Evaluating source code summarization techniques: Replication and expansion," in *2013 21st International Conference on Program Comprehension (ICPC)*. IEEE, 2013, pp. 13–22.

[21] G. Sridhara, E. Hill, D. Muppaneni, L. Pollock, and K. Vijay-Shanker, "Towards automatically generating summary comments for java methods," in *Proceedings of the IEEE/ACM international conference on Automated software engineering*, 2010, pp. 43–52.

[22] G. Sridhara, L. Pollock, and K. Vijay-Shanker, "Automatically detecting and describing high level actions within methods," in *2011 33rd International Conference on Software Engineering (ICSE)*. IEEE, 2011, pp. 101–110.

[23] L. Moreno, J. Aponte, G. Sridhara, A. Marcus, L. Pollock, and K. Vijay-Shanker, "Automatic generation of natural language summaries for java classes," in *2013 21st International Conference on Program Comprehension (ICPC)*. IEEE, 2013, pp. 23–32.

[24] E. Wong, T. Liu, and L. Tan, "Clocom: Mining existing source code for automatic comment generation," in *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE, 2015, pp. 380–389.

[25] P. W. McBurney and C. McMillan, "Automatic source code summarization of context for java methods," *IEEE Transactions on Software Engineering*, vol. 42, no. 2, pp. 103–119, 2015.

[26] P. Rodeghero, C. Liu, P. W. McBurney, and C. McMillan, "An eye-tracking study of java programmers and application to source code summarization," *IEEE Transactions on Software Engineering*, vol. 41, no. 11, pp. 1038–1054, 2015.

[27] Y. Lu, Z. Zhao, G. Li, and Z. Jin, "Learning to generate comments for api-based code snippets," in *Software Engineering and Methodology for Emerging Domains*. Springer, 2017, pp. 3–14.

[28] Y. Liang and K. Zhu, "Automatic generation of text descriptive comments for code blocks," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 32, no. 1, 2018.

[29] X. Hu, G. Li, X. Xia, D. Lo, and Z. Jin, "Deep code comment generation," in *2018 IEEE/ACM 26th International Conference on Program Comprehension (ICPC)*. IEEE, 2018, pp. 200–20 010.

[30] Y. Wan, Z. Zhao, M. Yang, G. Xu, H. Ying, J. Wu, and P. S. Yu, "Improving automatic source code summarization via deep reinforcement learning," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, 2018, pp. 397–407.

[31] U. Alon, S. Brody, O. Levy, and E. Yahav, "code2seq: Generating sequences from structured representations of code," in *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net, 2019. [Online]. Available: https://openreview.net/forum?id=H1gKYo09tX

[32] P. Fernandes, M. Allamanis, and M. Brockschmidt, "Structured neural summarization," in *International Conference on Learning Representations*, 2019. [Online]. Available: https://openreview.net/forum?id=H1ersoRqtm

[33] Q. Wang, B. Li, T. Xiao, J. Zhu, C. Li, D. F. Wong, and L. S. Chao, "Learning deep transformer models for machine translation," in *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*. Florence, Italy: Association for Computational Linguistics, Jul. 2019, pp. 1810–1822. [Online]. Available: https://www.aclweb.org/anthology/P19-1176

[34] J. Zhang, X. Wang, H. Zhang, H. Sun, and X. Liu, "Retrieval-based neural source code summarization," in *2020 IEEE/ACM 42nd Interna-*

tional Conference on Software Engineering (ICSE). IEEE, 2020, pp. 1385–1397.

[35] X. HU, G. LI, X. XIA, D. LO, S. LU, and Z. JIN, "Summarizing source code with transferred api knowledge.(2018)," in *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence (IJCAI 2018), Stockholm, Sweden, 2018 July 13*, vol. 19, pp. 2269–2275.

[36] B. Wei, G. Li, X. Xia, Z. Fu, and Z. Jin, "Code generation as a dual task of code summarization," in *Advances in Neural Information Processing Systems*, 2019, pp. 6563–6573.

[37] K. Xu, L. Wu, Z. Wang, Y. Feng, M. Witbrock, and V. Sheinin, "Graph2seq: Graph to sequence learning with attention-based neural networks," *arXiv preprint arXiv:1804.00823*, 2018.

[38] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," *arXiv preprint arXiv:1609.02907*, 2016.

[39] Y. Zhuang, Z. Liu, P. Qian, Q. Liu, X. Wang, and Q. He, "Smart contract vulnerability detection using graph neural network," in *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI-20*, C. Bessiere, Ed. International Joint Conferences on Artificial Intelligence Organization, 7 2020, pp. 3283–3290, main track. [Online]. Available: https://doi.org/10.24963/ijcai.2020/454

[40] "Solidity — solidity 0.6.0 documentation," https://docs.soliditylang.org/en/v0.6.0/index.html, 01 2021, (Accessed on 01/09/2021).

[41] "federicobond/solidity-parser-antlr: A solidity parser for js built on top of a robust antlr4 grammar," https://github.com/federicobond/solidity-parser-antlr, 01 2021, (Accessed on 01/26/2021).

[42] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[43] W. Wang, Y. Zhang, Y. Sui, Y. Wan, Z. Zhao, J. Wu, P. Yu, and G. Xu, "Reinforcement-learning-guided source code summarization via hierarchical attention," *IEEE Transactions on Software Engineering*, 2020.

[44] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, "Bleu: a method for automatic evaluation of machine translation," in *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*, 2002, pp. 311–318.

[45] C.-Y. Lin, "Rouge: A package for automatic evaluation of summaries," in *Text summarization branches out*, 2004, pp. 74–81.

[46] M. Denkowski and A. Lavie, "Meteor universal: Language specific translation evaluation for any target language," in *Proceedings of the ninth workshop on statistical machine translation*, 2014, pp. 376–380.

[47] B. Chen and C. Cherry, "A systematic comparison of smoothing techniques for sentence-level bleu," in *Proceedings of the Ninth Workshop on Statistical Machine Translation*, 2014, pp. 362–367.

[48] S. Stapleton, Y. Gambhir, A. LeClair, Z. Eberhart, W. Weimer, K. Leach, and Y. Huang, "A human study of comprehension and code summarization," in *Proceedings of the 28th International Conference on Program Comprehension*, 2020, pp. 2–13.