

CoSS: leveraging statement semantics for code summarization

Chaochen Shi, Borui Cai, Yao Zhao, Longxiang Gao, *Senior Member, IEEE*, Keshav Sood, and Yong Xiang, *Senior Member, IEEE*

Abstract—Automated code summarization tools allow generating descriptions for code snippets in natural language, which benefits software development and maintenance. Recent studies demonstrate that the quality of generated summaries can be improved by using additional code representations beyond token sequences. The majority of contemporary approaches mainly focus on extracting code syntactic and structural information from abstract syntax trees (ASTs). However, from the view of macro-structures, it is challenging to identify and capture semantically meaningful features due to fine-grained syntactic nodes involved in ASTs. To fill this gap, we investigate how to learn more code semantics and control flow features from the perspective of code statements. Accordingly, we propose a novel model entitled CoSS for code summarization. CoSS adopts a Transformer-based encoder and a graph attention network-based encoder to capture token-level and statement-level semantics from code token sequence and control flow graph, respectively. Then, after receiving two-level embeddings from encoders, a joint decoder with a multi-head attention mechanism predicts output sequences verbatim. Performance evaluations on Java, Python, and Solidity datasets validate that CoSS outperforms nine state-of-the-art (SOTA) neural code summarization models in effectiveness and is competitive in execution efficiency. Further, the ablation study reveals the contribution of each model component.

Index Terms—Code summarization, attention mechanism, control flow graph, graph neural network

1 INTRODUCTION

Code summarization employs natural sentences to describe the function of a source code snippet briefly [1, 2]. Concise and readable code summaries can be used as high-level comments to help developers understand, reuse and maintain source code, and thus can improve development efficiency significantly. Traditional code summarization is a labour-intensive manual task with a high error rate and difficulty in maintenance [3, 4]. As a result, automatic code summarization prevails and has been considered as a promising alternative [5].

The earliest research work on automatic code summarization dates back to Haiduc et al. [2, 6], who first attempted to generate code summaries using information retrieval (IR) techniques. After that, researchers concentrated mostly on rule- and IR-based heuristic approaches by extracting key information from code for sentence synthesis. In recent years, deep learning-based code summarization (DCS) has gradually become the mainstream due to its outstanding effectiveness. Typical DCS approaches, e.g., the representative CODE-NN [7] approach, are developed on the basis of machine translation and text summarization and follow an encoder-decoder architecture. They flatten structural source codes as plain texts and employ sequential models to generate code summaries. However, the structure

of source codes is critical to understanding their functionality [8], and represents essential semantics for code summarization tasks.

To conveniently represent code structures, recent studies adopt extra modalities, mostly Abstract Syntax Tree (AST) [9, 10, 11, 12, 13], for code summarization. AST represents the structure of the source code as a syntax tree, in which each node denotes a syntactic-level code construct. Such approaches demonstrate that adopting AST instead of plain texts improves model performance. However, AST breaks the basic unit (i.e., statement) of code into fine-grained syntax nodes, which usually results in AST being unnecessarily large and complicated with verbose attributes and repetitive mentions [14]. Code property graph is another code representation obtained by merging its AST, control flow graph (CFG), and program dependence graphs (PDG). CPG is more comprehensive while inherits shortcomings of AST. As a result, it is difficult and resource-consuming to extract semantically meaningful information from such code representations. Additionally, the complex structure of code representations also impairs the interpretability of captured features and the model reliability. Due to these limitations, we consider learning additional code semantics from other code representations.

Attention-based approaches are recently becoming very popular [13, 15] due to their excellent performance and model interpretability. We considered the way the attention mechanism works in code summarization models: Similar to human reading habits, an attention-based model focuses primarily on key tokens and statements. Code statements are the basic functional units of code, and they express concrete actions to be carried out. Besides, code statements can be structured as control flow, which represents program

- Chaochen Shi, Borui Cai, Yao Zhao, Keshav Sood, and Yong Xiang are with the School of Information Technology, Deakin University, Geelong, Australia.
E-mail: {shicha, b.cai, cnr, keshav.sood, yong.xiang}@deakin.edu.au
- Longxiang Gao is with Qilu University of Technology (Shandong Academy of Sciences).
E-mail: gaolx@sdaas.org

(Corresponding author: Yong Xiang)

execution paths (e.g., branches and loops). Our hypothesis is that structural information at the statement-level is easier to be understood than the verbose AST. Compared with the fine-grained syntax nodes of AST, statements and control flows are more effective in refining code functionality [3, 16, 17]. To capture the semantics of statements and their structures, we adopt control flow graph (CFG) as the statement-level representation of code. CFG nodes refer to statements, while edges between nodes refer to control flows. CFG represents code structure information at the statement-level, which could be more easily captured by attention-based models. This leads us to consider using CFG instead of AST as the extra input modality. Although CFG is widely used in program static analysis and vulnerability detection [18, 19], it is rarely explored in code summarization. For CFG, the semantics of each statement includes two parts, i.e., the code tokens it contains and the relations with neighboring statements. Thus, the main challenge of adopting CFG in code summarization is to jointly learn the textual semantics of individual statements and control flow features from their relationships with neighbor statements.

To break through this challenge, in this paper, we propose a novel code summarization approach named Code Summarization with Statement Semantics (CoSS)¹. CoSS follows the encoder-decoder architecture, where the encoders learn code semantics at token and statement levels, and the decoder receives code embeddings for code summary generation. Specifically, CoSS leverages two parallel encoders to learn two levels of semantics from two modalities – code token sequence and CFG, respectively. On the one hand, CoSS uses the Transformer-based [20] encoder to encode code token sequence and capture its text sequential features in token-level. On the other hand, we propose a stacked encoder to address the aforementioned challenge in learning statement-level textual semantics and control flow features from CFG. First, the semantics of statements are captured by a Bi-LSTM network, the output of which is adopted as the initial embedding of CFG nodes. Then, a Graph Attention Network (GAT) [21] is employed to capture structural information among statements by aggregating information of neighboring nodes/statements. Finally, a joint decoder takes the outputs from two encoders to generate code summaries. With multi-head attention, the joint decoder can dynamically capture important tokens and statements to improve the quality of generated summaries.

The main contributions of this paper are listed as follows:

- We investigate how to generate code summaries by learning comprehensive code semantics and structural aspects from the perspective of code statements, which is more effective in refining code functionality than the widely used AST.
- We design a novel approach entitled CoSS for code summary generation. Specifically, we develop a node2vec module and a GAT-based encoder for CoSS to achieve the joint extraction of textual and structural features from CFG. Moreover, we leverage a multi-head attention mechanism to unify the encoding and decoding process.

- We conduct comparative experiments on three programming languages (Java, Python, Solidity) datasets to evaluate CoSS. The experimental results show that CoSS outperforms SOTA DCS approaches on four widely-used metrics. Furthermore, an ablation study reveals the rationales behind our design.

2 PRELIMINARIES

In this section, we present background knowledge of adopted techniques including language model, Transformer, and graph encoder. We first introduce some basic notations and terminologies. Given a code snippet $\mathbf{x} = (x_1, x_2, \dots, x_{|\mathbf{x}|})$ of a function, the goal of code summarization is to generate a sequence of words, denoted as $\mathbf{y} = (y_1, y_2, \dots, y_{|\mathbf{y}|})$, from a dictionary as the high-level code comments, where x_i represents the i -th code token. For example, x_2 is the second token *function* in Python statement *def function(input);*; y_i represents the i -th generated words; $|\cdot|$ represents the length of \mathbf{x} or \mathbf{y} .

2.1 Language Model

In seq2seq tasks, language models compute the occurrence probability of each word in the generated sentences. The probability of a generated sentence with T words is denoted as $p(y_{1:T})$. Classic language models [22] usually compute $p(y_{1:T})$ based on the conditional probability of n-gram [23] predecessor words as shown in equation (1).

$$p(y_{1:T}) = \prod_{i=1}^{i=T} p(y_i | y_{1:i-1}) \approx \prod_{i=1}^{i=T} p(y_i | y_{i-(n-1):i-1}) \quad (1)$$

Such n-gram-based language models can not work properly, when there are n-grams never seen before. Moreover, those models are derived from counts of term co-occurrences, which leads to limited ability in semantic abstracting [24]. Fortunately, neural models can fill these gaps by leveraging neural networks with high abstracting power. Neural models used in seq2seq tasks follow encoder-decoder architectures. More specifically, the encoder transforms an input sequence \mathbf{x} of variable length into a vector v of fixed length. The hidden state h is initialized with v and a start symbol, then updates after reading each generated word y . The hidden state h_t at timestep t is computed as:

$$h_t = f(h_{t-1}, y_t), \quad (2)$$

where f is usually an recurrent network like RNN [10, 25]. Then, the decoder predicts the subsequent word based on h_t as:

$$p(y_{t+1} | y_{1:t}) = g(h_t), \quad (3)$$

where g is a stochastic output layer, e.g., a softmax layer, which outputs the probability of selecting each word.

1. Replication package is at <https://github.com/PatrickNinja/CoSS>

2.2 Transformer

The commonly used RNN- and LSTM-based language models are exposed with the challenge of long-distance dependency. Specifically, RNN and LSTM are calculated sequentially, and thus they take many time steps to accumulate information for the connection of two long-distance inter-dependent features. This may cause the vanishing gradient problem (LSTM mitigates this problem in a time-consuming way). Transformer fills this gap by leveraging self-attention mechanisms. In the calculation process of self-attention, any two words in a sentence are directly connected through one step, greatly shortening the distance between inter-dependent features. In addition, self-attention mechanisms allow parallel computing, which greatly improves model efficiency. Transformer-based models, e.g., BERT [26], XLNET [27], and their variants have been proven to be SOTA models in many seq2seq tasks, especially code summarization. Thus, we choose Transformer as the backbone of our approach. We will not introduce more details of Transformer here, as it is well-known and widely used. Our design is based on an original transformer architecture and ignores its variants so as to focus on its design philosophy.

2.3 Graph Encoder

Different from normal texts, a program can be represented as a graph such as AST or CFG. In this case, graph-specific encoders are required. A widely-used method to embed a graph is to flatten it into a sequence through traversal, then feed the result into sequence models such as RNN and LSTM [9, 25]. Nevertheless, traversal causes unrecoverable information loss, especially edge features. Thus, we prefer using graph neural networks (GNNs) as encoders for graphs. Compared with Multi-layer Perceptron (MLP), GNN adds an adjacent matrix which indicates the graph structural information as an additional element. GNN can be formulated as:

$$H = \alpha(AXW), \quad (4)$$

where H denotes an output state, α is an activation function, A is an adjacent matrix, X is a node feature matrix, and W represents node weights. Graph convolutional network (GCN) [28] is a representative GNN which expands convolution operation from classic data (e.g., images) to graph data. Its core idea is to perform the first-order approximation on the convolution kernel around each node. As a transductive model, GCN can not embed graphs that are never seen in the training stage and is not applicable to be adopted in directed graphs. GAT introduces a self-attention mechanism on the basis of GCN to address this limitation. GAT learns the importance (attention) of neighbor nodes to the target node. This allows GAT to obtain the SOTA performance on public datasets without the need for matrix calculations or prior knowledge of the graph structure [21]. Thus, GAT is more applicable for graph-to-sequence tasks than GCN and is able to handle directed graphs, e.g., CFG.

3 ILLUSTRATIVE EXAMPLE

We employ a Solidity code snippet shown in Listing 1 as an example to illustrate the attention mechanism used in CoSS.

Figure 1 illustrates the attention heatmap between encoders and the decoder regarding Listing 1, where the middle part is the ideal summary generated by the decoder, and the left and the right parts are the outputs of self-weighted code token sequence encoders and CFG encoders, respectively. Taking the prediction of the word “*deposits*” as an example, it pays more attention to the token “*balance*” from the code token sequence, as they are synonyms. Besides, it focuses more on statement 2 which further defines the semantic of “*balance*” through an assignment. The transformer-based encoder captures long-distance dependencies among code tokens through a self-attention mechanism, thus injecting the global semantic information to code tokens. In contrast, the GAT-based encoder extracts local semantic information between neighbouring statements from CFG. As CFG represents the execution paths of a program, local semantic information extracted from it contains significant code structural features, e.g., branches and loops. For example, both statements 4 and 6 aggregate 1-hop neighbor information from statement 3 and the corresponding branch information, i.e., true and false.

```
function withdraw() public returns (uint){ 1
    uint amount = balance[msg.sender]; 2
    if (amount > 0) { 3
        deposits[msg.sender] = 0; 4
        msg.sender.transfer(amount); 5
    }
    emit withdrawDone(amount, balance[msg.sender], msg.sender); 6
    return amount; 7
}
```

Code Listing 1: Withdraw function in Solidity. It aims to withdraw the rest of deposits from the user account.

4 METHODOLOGY

In this section, we articulate our proposed CoSS. Overall, our approach follows an encoder-decoder architecture, where two encoders learn token-level and statement-level semantics from token sequences and CFGs, respectively. Then, a joint decoder integrates two outputs from encoders and produces code summaries in natural language.

4.1 Code Embedding and Encoders

The architecture of code encoders is shown in Fig. 2. Specifically, we adopt two encoders: a transformer-based encoder and a GAT-based encoder for extracting text semantics from code token sequences and for extracting structural semantics from CFG, respectively.

4.1.1 Code Token Embedding and Encoder

Due to the outstanding performance of Transformer [20] in seq2seq tasks, we reuse Transformer’s encoder to embed code tokens. Original code texts, as a bag of words (tokens), are split by a set of symbols, i.e., $\{ . , ' : () ! - (\text{space}) \}$. Each token written in camel-case or snake-case shall be segmented into original words. For example, the token *helloWorld* or *hello_world* shall be segmented into two separate words: *hello* and *world*. Then, Word2Vec [29] is adopted to generate the initial vector representation $V_X = \{\vec{v}_{x_1}, \vec{v}_{x_2}, \dots, \vec{v}_{x_t}\}$ corresponding to the token sequence $X = \{x_1, x_2, \dots, x_t\}$, where v_{x_i} is the $1 \times F$ vector representation of the i -th token and t is the maximal length

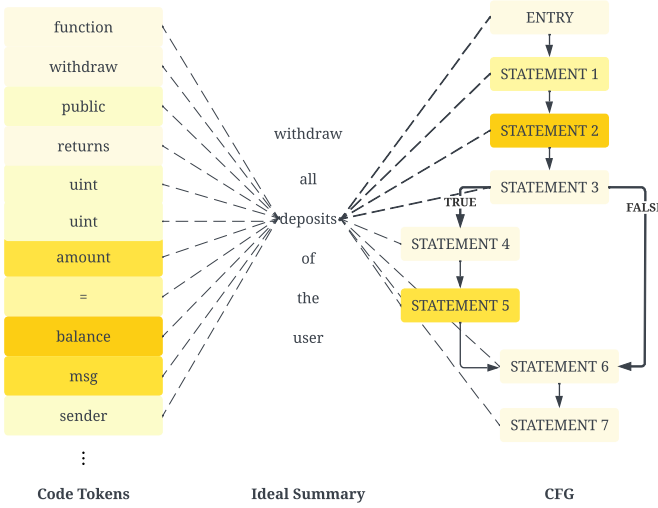


Fig. 1: The co-attention weight to the token “deposits” in line 2 of Listing 1. Darker colors represent higher weights.

of token sequences. For sequences shorter than t , we pad the remaining part with zeros and mask them out. Positional encoding is also applied to inject relative order information into token vectors. Next, similar to words processed in Transformer, token vectors are fed into a multi-head self-attention layer and a point-wise feed-forward layer. Each layer has an Add&Norm sub-layer with a residual connection. The multi-head self-attention layer outputs token vectors with self-attention-weights that are used to identify which tokens are more important to a certain token in semantics. Finally, the feed-forward layer adjusts and outputs an embedding vector collection $V'_X = \{\vec{v}'_{x_1}, \vec{v}'_{x_2}, \dots, \vec{v}'_{x_t}\}$.

4.1.2 CFG Embedding and Encoder

As mentioned in section 1, the main challenge of adopting CFG is learning textual semantics while aggregating control flow features of statements. To address it, we design two specific structures, including a node2vec module and a GAT-based encoder. The former is adopted to generate semantic features for each CFG node, while the latter is used to extract structural features of the whole CFG.

The node2vec module is illustrated in Fig. 3. It receives the node collection $N = \{n_1, n_2, \dots, n_k\}$ of length k and outputs a corresponding vector representation $V_N = \{\vec{v}_{n_1}, \vec{v}_{n_2}, \dots, \vec{v}_{n_k}\}$, where n_i refers to the i -th node. Since each statement of source code is a node in CFG, and a statement is a sequence of code tokens, we reuse a Word2vec module and add a Bi-LSTM layer to generate the initial embeddings of nodes. Supposing the statement starts from the l -th token of \mathbf{x} , the Word2vec module slides across m tokens and generates the $m \times F$ vector matrix $[\vec{v}_{x_l}, \vec{v}_{x_{l+1}}, \dots, \vec{v}_{x_{l+m}}]$ as the vector representation of the statement. The relative position of words in a statement contains important semantic information, and thus we feed the vector matrix to the Bi-LSTM layer to capture sequential dependencies between words in a statement and to generate a $1 \times F$ vector as the initial embedding of the node.

There are three types of edges in CFG: **Natural Sequence (NS)**, **True**, and **False**. NS edges represent natural execution

orders of statements while **True** and **False** edges connect branches. **True** and **False** edges are converted to nodes with identifier $\langle T \rangle$ and $\langle F \rangle$, respectively, connected with neighbors through NS edges. In this way, CFG is converted to a directed graph with single-type edges, which can be handled by GAT directly. Moreover, CFG reserves a node pointing to the entry block of the program. We label this node with identifier $\langle E \rangle$. The nodes with special identifiers are also embedded as $1 \times F$ vectors. Therefore, the output V_N of the node embedding module can also be merged as a vector matrix of size $k \times F$, where each row of the matrix corresponds to the initial embedding of a CFG node. A graph neural network is adopted as the encoder to aggregate control flow information into the initial node embeddings. Specifically, GAT captures relationships between CFG nodes, i.e., structural features of CFG. GAT uses an attention mechanism to learn new node representations by aggregating the features of neighbors. Compared with the Laplacian matrix used in graph convolutional networks (GCNs), the attention matrix can better capture correlation information between neighbors. Additionally, GAT can be directly employed in inductive tasks and directed graphs, which fits our scenarios well.

The input to the graph attention layer is the output V_N from the node embedding module. We perform masked attention by only computing α_{ij} for nodes $j \in \mathcal{N}_i$, where \mathcal{N}_i is the collection of node i 's first-order neighbors identified through the adjacent matrix of CFG. The normalized attention coefficient α_{ij} between node i, j are calculated as equation (5).

$$\alpha_{ij} = \frac{\text{LeakyReLU}(\vec{a}^T [W\vec{v}_{n_i} \| W\vec{v}_{n_j}])}{\sum_{k \in \mathcal{N}_i} \text{LeakyReLU}(\vec{a}^T [W\vec{v}_{n_i} \| W\vec{v}_{n_k}])}, \quad (5)$$

where \vec{a}^T is a $2F \times 1$ attention kernel, W represents an input linear transformation's weight matrix, and $\|$ denotes a concatenation operation. Similar to Transformer, We adopt multi-head self-attention to stabilize the learning process of the model. The aggregated features from each head are concatenated to obtain \vec{v}''_{n_i} .

$$\vec{v}''_{n_i} = \left\|_{k=1}^K \sigma \left(\sum_{k \in \mathcal{N}_i} \alpha_{ij}^k W^k \vec{v}_{n_j} \right) \right\|, \quad (6)$$

where K is the number of heads, σ is an exponential linear unit (ELU) [30], α_{ij}^k and W^k are α_{ij} and W of the k -th head, respectively. Then, \vec{v}''_{n_i} is fed to a feed-forward layer (FFN) and an Add&Norm layer to obtain the final representation \vec{v}'_{n_i} of the i -th node.

$$\vec{v}'_{n_i} = \text{LayerNorm}(\vec{v}''_{n_i} + FFN(\vec{v}''_{n_i})) \quad (7)$$

In this way, we obtain the output of CFG embedding module: $V'_N = \{\vec{v}'_{n_1}, \vec{v}'_{n_2}, \dots, \vec{v}'_{n_k}\}$, $\vec{v}'_{n_i} \in \mathbb{R}^F$.

4.2 Joint Decoder

We design a joint decoder as shown in Fig. 4 to receive hidden space vectors V'_X and V'_N from code encoders, and then decode them to natural language comments, i.e., code summaries. The decoding process starts from a $\langle SOS \rangle$ tag and shifts right word by word. The prediction of the word at

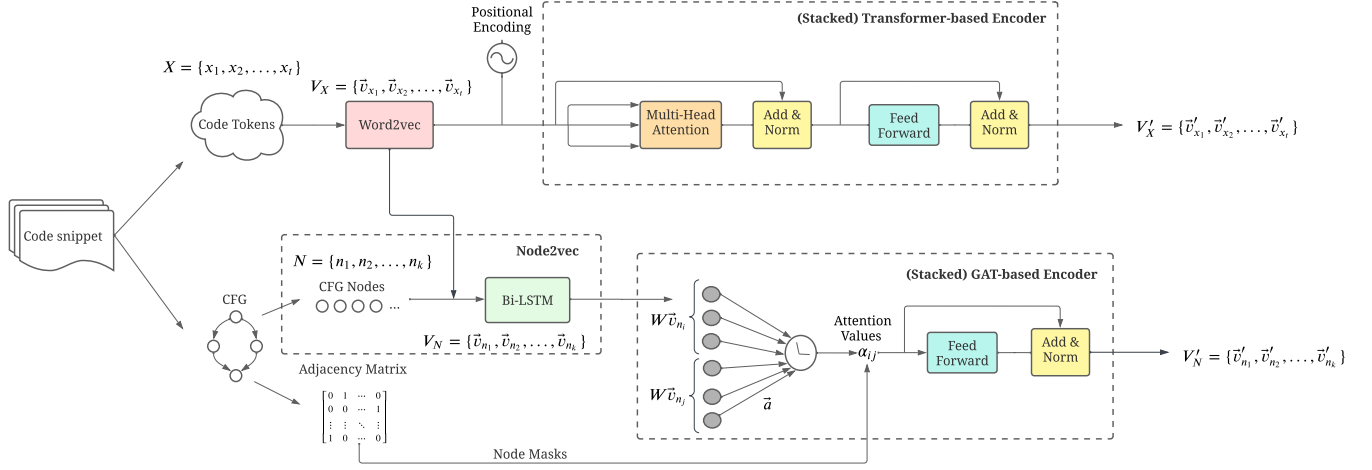


Fig. 2: The architecture of code embedding in our approach. It contains two parts to learn semantic feature representation of source code. The top half follows the Transformer structure, aiming to embed code tokens with the self-attention mechanism. The bottom half aims to extract semantic features from CFG through a node embedding module and a GAT-based encoder.

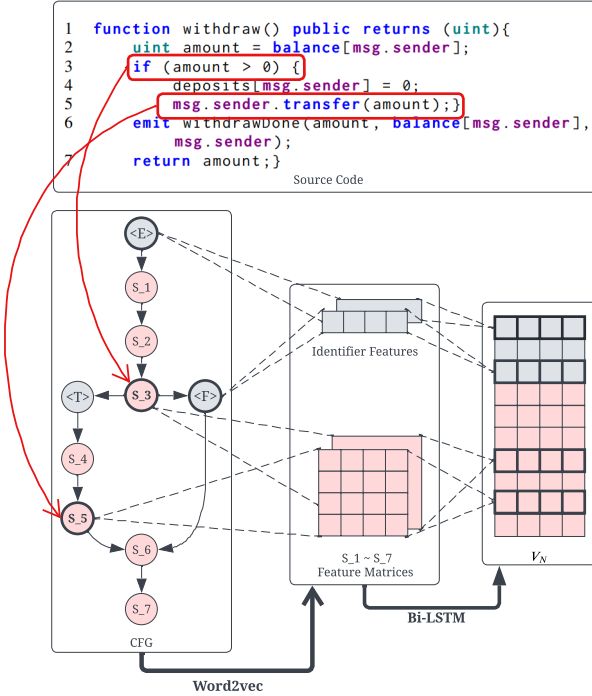


Fig. 3: The overall structure of node2vec module. Semantic features of each CFG node are embedded as a vector of size $1 \times F$ and together form a vector matrix V_N of size $k \times F$. k is the number of nodes. S_N is the N -th code statement.

timestep t depends on the words generated from timestep 0 to $t - 1$. Similar to the code tokens embedding process, the joint decoder embeds generated words by a Word2vec layer and injects positional information, and then feeds them to a multi-head self-attention module to extract self-attention correlations. However, ground truth data is processed as a whole sentence during the training, leading to the information leakage of the words after timestep $t - 1$. Thus, we leverage an upper triangular matrix to mask the words after

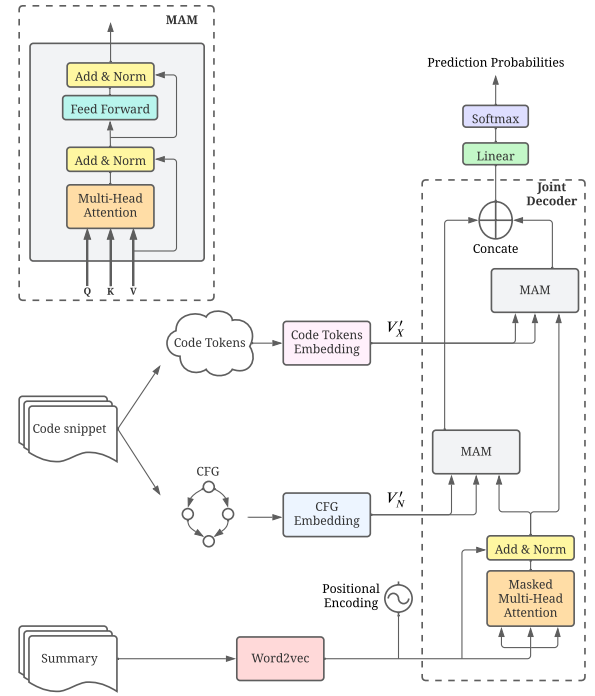


Fig. 4: The architecture of code summarization model

the current timestep.

Then, we adopt two individual multi-head attention modules (MAMs) to capture encoder-decoder attention information. One MAM receives V'_X as the input Q, K , where V derives from the generated $t - 1$ words. Another MAM handles V'_N in the same way. The rationale behind this step is to allow the decoder to predict the word at timestep t by selecting more important code tokens and CFG nodes (code statements). The outputs from two MAMs are then concatenated on their last axis to merge their predictions. Finally, the concatenated output is transformed into the prediction probability of the next word through a linear

layer and a softmax layer.

The training objective is to maximize the probability of generating the target word y with the input, and thus the loss is defined as:

$$Loss = -\frac{1}{|y|} \sum_{t=1}^{|y|} \log(P(\hat{y}_t)), \quad (8)$$

where $P(\hat{y}_t)$ is the probability that the t -th token of the y is predicted as the ground truth. A teacher forcing strategy [31] is employed to enable the parallel computation of prediction losses in the training stage.

5 EXPERIMENTS AND ANALYSIS

We conduct extensive experiments to show the superiority of CoSS and its core components. The experiments focus on the following research questions:

- **RQ1:** What is the performance of CoSS compared with SOTA approaches on different programming languages?
- **RQ2:** How do the lengths of code snippets affect the model performance?
- **RQ3:** How much do different modalities and components contribute to CoSS?
- **RQ4:** What is the training efficiency of CoSS?

All experiments are implemented with Python 3.7 and run on an NVIDIA Tesla P100 GPU. The number of attention heads of both Transformer- and GAT-based encoders in CoSS is set to 8, which is the widely used default setting in many tasks. The mini-batch size is set to 32, and the learning rate is 0.001, following settings in [25].

5.1 Dataset

We evaluate CoSS and SOTA approaches on Java, Python, and a representative domain-specific programming language named Solidity. Solidity is a statically-typed curly-braces programming language designed for developing smart contracts that run on Ethereum. Solidity has many unique features, such as transaction triggers and blockchain-related terms. In addition to Java and Python, we would like to know the performance of CoSS on such domain-specific programming languages. The analysis of the experimental results facilitates our subsequent optimization of CoSS for more domain-specific languages. We adopt the Java project dataset from CodeSearchNet [32] which contains 496K Java $\langle code, comment \rangle$ pairs. For Python, we employ the dataset provided by Barone [33], which is widely adopted in evaluating DCS approaches [10, 25, 34]. After removing records with incomplete code snippets and comments, there are 108K $\langle code, comment \rangle$ pairs left. For Solidity, we adopt the dataset in [35] which contains 48,578 pairs. Note that all collected code snippets are complete functions. We split original code texts by a set of symbols, i.e., $\{., ", ' : () ! - (space)\}$. We also collect graph representations of code snippets: ASTs are from original compilers, and CFGs are produced by third-party tools, e.g., *slither* [36]. We randomly divide the Solidity dataset into training, test, and validation sets in proportion with 8: 1: 1 and run 10 epochs in training processes.

The length of a code snippet is measured by the number of its tokens. The length distributions of the washed code snippets are shown in Fig. 5. We can observe that most of the code snippets are within 5-100 tokens. Moreover, we notice that almost all the comments in our dataset are within 5-20 tokens. It indicates that the generated comments should usually be short sentences.

5.2 Evaluation Metrics

We evaluate the performance of DCS models based on four metrics that are commonly used in code summarization tasks, i.e., BLEU [37], ROUGE-L [38], METEOR [39], and CIDER [40]. Specifically, BLEU measures the n-gram precision (from unigrams to 4-grams in practice) on a set of reference sentences, with a penalty for overly short sentences. As comments are usually short, higher-order n-grams may not overlap, and thus we adopt a smoothed BLEU-4 score and the same evaluation scripts as used in [7]. Another metric named ROUGE-L matches the longest common sequence between two sentences and returns recall rate. METEOR considers both accuracy and recall rate, and returns F value. CIDER is a combination of BLEU and vector space model, which evaluates the cosine similarity between generated sentences and references based on their TF-IDF vectors. The scores of BLEU, METEOR, and ROUGE-L in percentages since they are in the range of [0, 1]. As CIDER scores range in [0, 10], we display them in real values. A higher number indicates better quality of generated sentences.

5.3 RQ1: Performance comparison between CoSS and SOTA approaches

We compare CoSS with nine SOTA DCS approaches on three datasets. The selected approaches are listed in Table 3. In particular, CODE-NN is the first DCS approach that uses attention-based LSTM to produce code summaries that describe C# code snippets and SQL queries. Code2seq represents a code snippet as the set of compositional paths in its AST and employs attention to select relevant paths while decoding. Both Hybrid-DRL and RL-Guided apply RL-based techniques (actor-critic network) in decoding, where Hybrid-DRL employs Tree-LSTM to embed AST, and RL-Guided represents graph modalities as sequences through traversal. Although RL-Guided adopts CFG as one of the input modalities, the encoding method is based on traversal and sequence modeling, which loses edge features. CoCoSum utilizes a UML graph as an additional inter-context and encodes it using a novel MRGNN. CAST splits AST into sub-trees and encodes them through RvNN, then aggregates the embeddings of sub-trees to obtain the representation of the complete AST. CodeBERT [15] is a Transformer-based model pre-trained on six different programming languages and fine-tuned on the datasets. It supports downstream programming language to natural language (PL-NL) applications, including code summarization. DFG is another code representation that reflects the global syntactic structure. SG-Trans [41] uses DFG as the input modality and achieves SOTA performance under the Transformer framework. We also consider CPG-based approaches as baselines because of the high comprehensiveness of CPG [42] in code representation. SmartDoc [43] adopts the Transformer

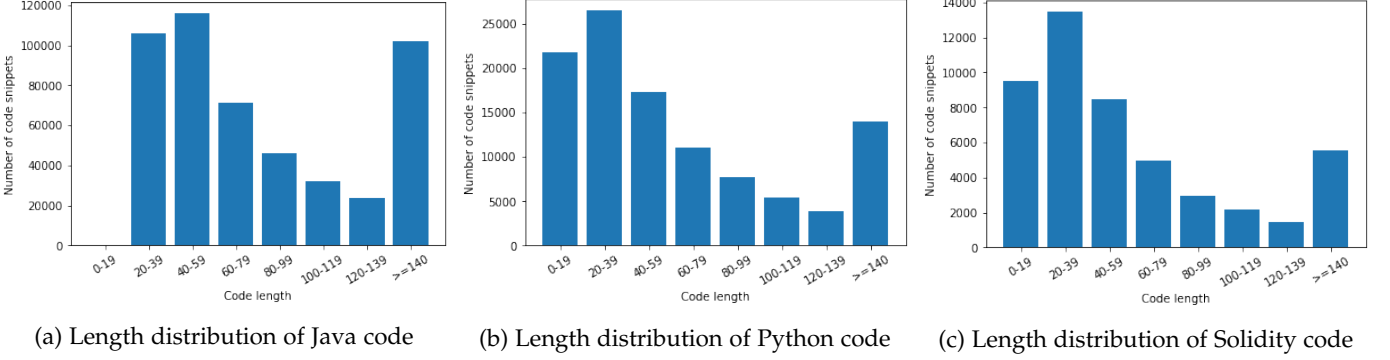


Fig. 5: Length distributions of code snippets (functions)

TABLE 1: Comparison of the overall performance between CoSS and baselines (in percentage). B: BLEU-4, M: METEOR, R: ROUGE-L, C: CIDER.

Model	Java				Python				Solidity			
	B	M	R	C	B	M	R	C	B	M	R	C
CODE-NN	19.33	18.04	26.77	0.58	12.95	5.85	18.47	0.44	10.42	4.57	16.65	0.37
Code2seq	13.25	10.07	23.56	0.45	16.32	6.73	23.35	0.61	15.16	6.98	21.00	0.55
Hybrid-DRL	25.11	9.29	39.13	0.75	14.95	8.37	22.01	0.65	14.39	7.87	24.76	0.73
RL-Guided	35.10	20.01	37.19	1.68	26.18	10.43	34.12	1.36	22.08	8.55	26.30	1.09
CoCoSum	30.01	16.04	37.83	1.46	21.13	13.06	30.48	1.27	18.64	10.88	26.65	0.97
CAST	32.15	22.78	35.59	1.72	24.89	12.62	27.31	1.28	21.78	10.85	23.94	0.98
SG-Trans	35.89	23.67	44.18	1.54	25.06	12.86	31.13	1.22	21.91	11.01	25.87	1.02
CodeBERT	19.89	17.33	29.48	1.04	28.58	11.54	35.65	1.40	11.12	5.50	15.48	0.43
SmartDoc	-	-	-	-	-	-	-	-	16.31	8.57	22.65	0.88
CoSS	36.70	25.74	40.41	1.79	30.26	14.64	41.01	1.40	26.59	12.07	32.34	1.21

TABLE 2: Comparison of the overall performance between CoSS and HGNN on CCSD.

Model	C		
	B	M	R
HGNN	14.01	14.50	30.89
CoSS	13.84	14.63	32.25

and Pointer mechanism to generate user notices of Solidity source code, to help users better understand smart contracts. Moreover, SmartDoc utilizes transfer learning to transfer the knowledge from Java methods to generate Solidity user notices by pre-training. HGNN [44] is the first and only CPG-based approach so far. Since the authors didn't release replication packages, we are not able to evaluate HGNN on our datasets. Thus, we train and evaluate CoSS on the C Code Summarization Dataset (CCSD) [44] which is a C benchmark built by the authors of HGNN, additionally.

Table 1 demonstrates the performance of CoSS and aforementioned baselines. The results reveal that CoSS outperforms baselines on three datasets. On the Java dataset, CoSS improved over the best baseline by 2.2%/8.7%/4.1% on BLEU-4, METEOR, and CIDER, respectively, and was second only to SG-Trans on ROUGE-L. For Python, CoSS achieves the best scores on four metrics and improves the best baselines by 5.9%/12.0%/15.0% on BLEU-4, METEOR, and ROUGE-L, respectively, and reaches the same CIDER score as CodeBERT. Similarly, CoSS upgrades the best baselines by 20.4%/9.6%/21.3%/8.0% on BLEU-4, METEOR, ROUGE-L, and CIDER on the Solidity dataset, although the performance of models drops due to the smaller training sets. In addition, Code-NN is the poorest model, since it only takes code tokens as input and fails to capture

TABLE 3: Representative and SOTA DCS approaches. Abbreviations: RL (Reinforcement Learning), TL (Transfer Learning), RvNN (Recursive Neural Network), MRGNN (Multi-Relational Graph Neural Network), HAN (Hierarchical Attention Network), UML (Unified Modeling Language), DFG (Data Flow Graph)

Approaches	Primary Technique	Modality
CODE-NN [7]	LSTM	Code Tokens
Code2seq [45]	LSTM	AST (Paths)
Hybrid-DRL [10]	Tree-LSTM, RL	Code Tokens, AST
RL-Guided [25]	HAN, RL	Code Tokens, AST, CFG
CoCoSum [46]	GRU, MRGNN	Code Tokens, AST, UML
CAST [13]	Transformer, RvNN	Code Tokens, AST (Split)
SG-Trans [41]	Transformer	Code Tokens, DFG
SmartDoc [43]	Transformer, Pointer, TL	Code Tokens
CodeBERT [15]	pre-trained Transformer	Code Tokens
HGNN [44]	Hybrid GNN, IR	CPG

deeper code features. CodeBERT shows good performance in Python because it has been pre-trained on a Python corpus CodeSearchNet [32]. We further observe that when CodeBERT runs on Solidity which it has never seen before, its performance drops significantly to the same level as CODE-NN. Since CodeBERT is pre-trained on CodeSearchNet, it also did not show superiority on the Java dataset collected from the same corpus. Moreover, RL-Guided achieves the best performance among all the AST-based baselines. Our CoSS innovatively utilizes the statement-level attention brought by CFG rather than the syntactic details brought by AST and achieves the best results, which demonstrates the superiority of our model.

Table 2 shows that CoSS and HGNN are quite close in performance on the CCSD dataset. However, the short-

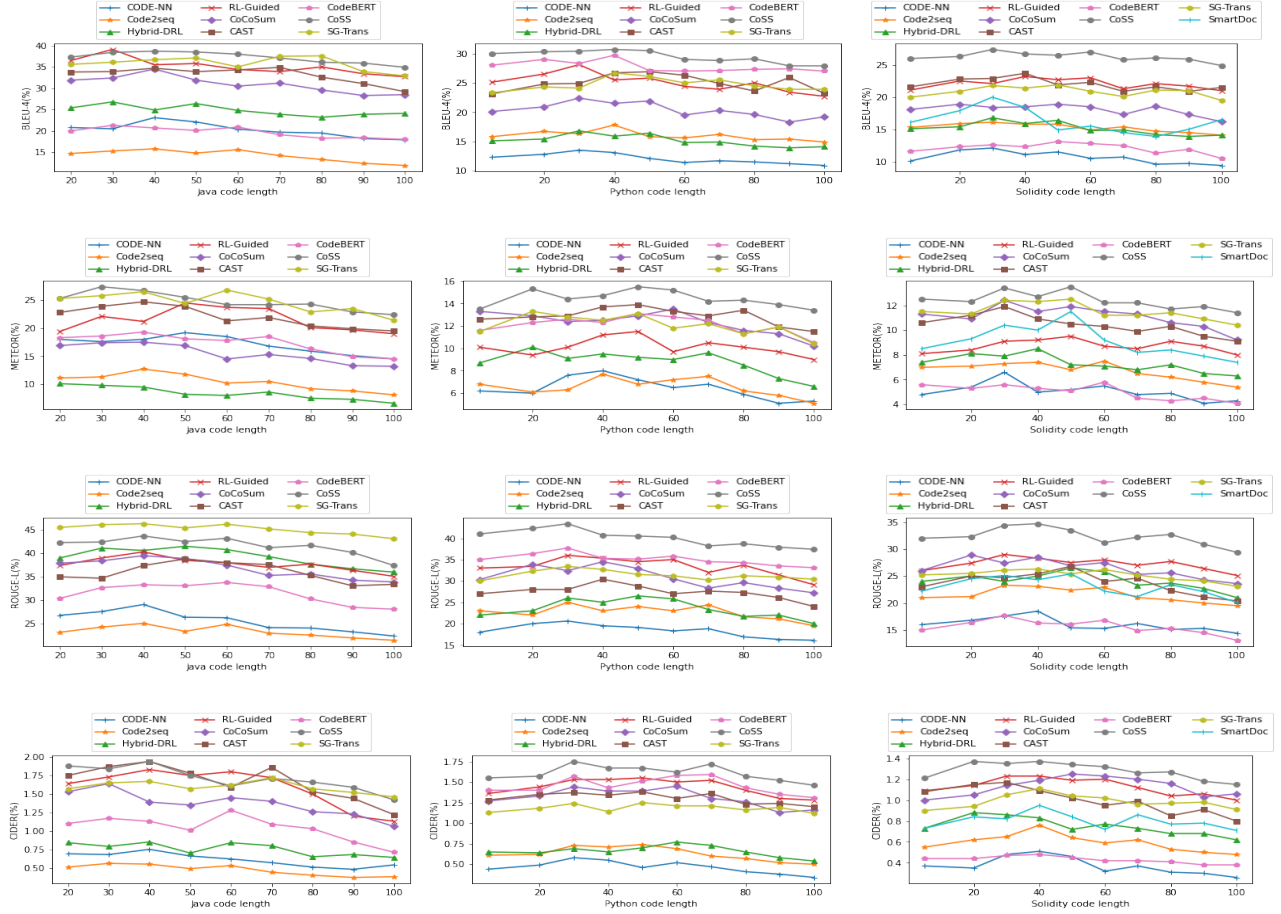


Fig. 6: Model performance under different code lengths

comings of using CPG are obvious: like AST, CPG is complicated, which causes high consumption of training time and hardware resources. Moreover, the existing code property graph generators only support a limited number of programming languages including C/C++, java, Kotlin, Python, JavaScript, LLVM bytecode, etc. Programming languages such as Solidity are not supported yet, which limits its application scenarios.

5.4 RQ2: Performance under different code lengths

Figure 6 shows the performance of CoSS and baselines under different code lengths. As the lengths of the majority of code snippets are ranged between 5 and 100, we evaluate the performance of models in this range. The three columns show the results in four metrics on Java, Python, and Solidity from left to right, respectively. We can observe that CoSS outperforms baselines under almost all the different code lengths. Besides, most of the models including CoSS achieve their best performance when code lengths are between 10 and 40, with a minor decline in performance as code length increases. The trends in the score changes with code lengths are similar, even though the same model scores differently on the python and solidity datasets. Nevertheless, the performance of CoSS remains stable overall, as the gaps between the maximum and minimum values of each metric are always within 20%.

5.5 RQ3: Effectiveness of each modality and component

We conduct an ablation study to evaluate the effectiveness of different modalities and components for CoSS. The performance of CoSS is evaluated under various combinations of three input modalities, i.e., code tokens, CFG, and AST. Code tokens and CFG are the original modalities of CoSS, while AST is widely used in SOTA approaches. We incorporate AST into the ablation study to verify the rationale behind our modalities selection, where the AST is embedded with GAT as well. The ablation study also focuses on four core components: Transformer, Bi-LSTM, GAT, and joint decoder.

- **CoSS without Transformer encoder (CoSS-T).** Similar to CODE-NN [7], this baseline takes LSTM rather than Transformer as the code tokens encoder in CoSS. The encoded hidden state is sent to the joint decoder for further decoding.
- **CoSS without Bi-LSTM (CoSS-B).** This baseline uses average-pooling, calculating the average for each token of the sentence instead of using Bi-LSTM to get an embedding of the statement (CFG nodes).
- **CoSS without GAT (CoSS-G).** This baseline conducts the classic deep-first traversal to flatten a CFG into a sequence and embed it with LSTM, similar to [25].

TABLE 4: CoSS performance under different modalities and components. B: BLEU-4, M: METEOR, R: ROUGE-L, C: CIDER.

Modality/Variant	Java				Python				Solidity			
	B	M	R	C	B	M	R	C	B	M	R	C
Code tokens	29.27	20.16	31.45	1.34	20.44	9.18	23.78	0.95	18.62	8.80	21.77	0.79
AST	22.13	13.52	26.80	0.92	14.32	5.53	19.35	0.53	12.11	4.98	17.04	0.49
CFG	24.12	12.57	30.11	1.05	18.83	7.79	22.55	0.80	17.41	7.08	20.56	0.70
Code tokens + AST	33.45	22.82	36.99	1.51	23.48	11.10	26.81	1.17	21.08	9.52	23.15	0.95
Code tokens + CFG (CoSS)	36.40	25.74	40.41	1.79	30.26	14.64	41.01	1.40	26.59	12.07	32.34	1.21
AST + CFG	31.05	23.73	33.00	1.22	22.81	10.45	23.99	1.08	19.78	9.85	22.94	0.88
Code tokens + AST+ CFG	36.91	25.07	42.06	1.72	30.65	14.17	39.83	1.37	25.88	12.97	34.23	1.15
CoSS-T	29.12	18.79	29.08	1.21	21.33	8.97	24.95	1.02	19.80	7.45	20.04	0.80
CoSS-B	34.44	23.26	37.14	1.66	27.35	13.57	33.13	1.22	23.42	10.01	29.25	1.26
CoSS-G	30.97	22.98	33.39	1.40	24.20	11.02	25.34	1.18	20.76	9.95	24.94	1.02
CoSS-J	28.09	15.45	27.61	0.97	18.68	8.59	20.17	0.64	16.61	8.10	19.04	0.60

- **CoSS without joint decoder (CoSS-J).** This baseline simply adopts LSTM as the decoder. The encoder outputs are concatenated as a single input vector of the decoder.

The results of the ablation study are shown in Table 4. As for modalities, we can observe that baselines with hybrid modalities can improve the model performance on all four metrics compared with uni-modal baselines. For the double-modalities combinations, baselines with code tokens + CFG (our approach) outperforms code tokens + AST, which verifies that CFG contributes more to the model performance than AST. Surprisingly, we find that the baseline with triple-modalities (code tokens + AST + CFG) has similar performance with code tokens + CFG. That indicates the combination of code tokens and CFG almost covers the code semantics extracted from AST. Theoretically, code tokens cover parts of AST nodes, and CFG covers parts of structural information represented by AST paths. The syntax details, such as variable types, are exclusive to AST while contributing little to summary generation. The underlying reason is that high-level descriptions normally do not involve such detailed information.

Table 4 also shows the effectiveness of each core component of CoSS. Transformer structure is the backbone of CoSS and is adopted in both the code token encoder and the joint decoder. Taking BLEU-4 scores on Python as an example, replacing Transformer with classic LSTM in the encoder and decoder would decrease the model performance by 29.5% and 38.2%, respectively. Another observation is that using Bi-LSTM to generate an initial embedding of code statements is better than using average pooling by 10.6%. This is because Bi-LSTM brings more learnable parameters, while the pooling method is a simple down-sampling operation. Additionally, GAT used in CFG encoder improves the traditional “traversal to sequence encoder” approach by 25.0%. It verifies that GAT can efficiently capture the structural features from CFG and make significant contributions to summary generation.

5.6 RQ4: Training efficiency

Table 5 shows the average time consumption per training epoch for CoSS and baselines on our datasets. We can observe that the training efficiency of CoSS is only second to the pre-trained model CodeBERT and SmartDoc among all SOTA models. The training for CodeBERT and SmartDoc are fine-tuned processes that are fast due to their uni-modal input and Transformer structure. Compared with traditional

TABLE 5: The average time consumption in training (min-s/epoch).

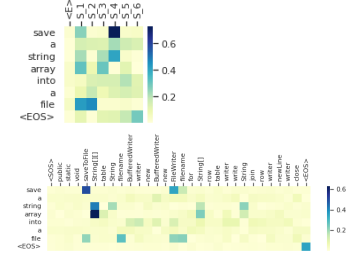
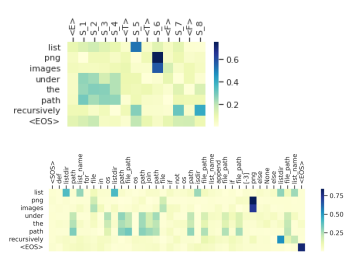
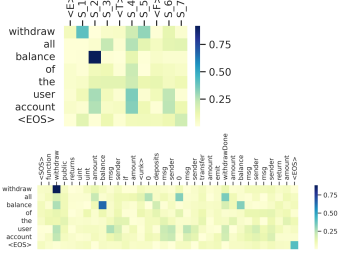
Model/Modality/Variant	Java	Python	Solidity
CODE-NN	95	24	14
Code2seq	101	27	15
Hybrid-DRL	178	44	24
RL-Guided	194	52	30
CoCoSum	171	48	27
CAST	110	31	18
SG-Trans	103	26	15
CodeBERT	71	19	11
SmartDoc	-	-	10
CoSS	83	21	13
Code Tokens	56	16	8
AST	94	21	11
CFG	45	11	6
Code tokens + AST	108	28	14
AST + CFG	99	26	13
Code tokens + AST + CFG	152	39	21
CoSS-T	40	95	23
CoSS-B	68	17	8
CoSS-G	80	22	13
CoSS-J	91	24	13

sequential models such as RNN and LSTM, Transformer allows efficient parallelization when encoding. CoSS also adopts Transformer in the code token encoder, which greatly promotes training efficiency. Besides, we can see that CoSS-T spends twice as much time in one epoch training as CoSS, while the time costs of other variants are similar to CoSS. It indicates that the Transformer-based encoder contributes more to the training efficiency of CoSS than other components. Moreover, when other conditions remain unchanged, models using CFG are significantly more efficient than models using AST in training. The main reason is that the structure of CFG is much simpler and easy to process. The gap between CFG- and AST-based models is more obvious on the python dataset than on the solidity dataset, which evidences that the gap increases as the amount of training data grows. According to the results, the model with code tokens + CFG (CoSS) achieves the best balance between performance and training speed.

5.7 Case study

Table 6 shows three cases. Obviously, summaries generated by CoSS are the most similar to the ground truths in all the cases. Although baselines without an attention mechanism and CFG modality can also generate recapitulative descriptions, they can hardly capture the branch- and loop-related features. For example, none of them captures the semantics of recursive operation in case 2 and generates the

TABLE 6: Code summaries generated by CoSS and baselines.

	Case 1 (Java)	Case 2 (Python)	Case 3 (Solidity)
Code Snippet	<pre>public static void saveToFile(String [][] table, String filename) { BufferedWriter writer = new BufferedWriter(new FileWriter(filename)); for (String[] row : table) { writer.write(String.join(", ", row)); writer.newLine(); } writer.close(); }</pre>	<pre>def listdir(path, list_name): for file in os.listdir(path): file_path = os.path.join(path, file) if not os.path.isdir(file_path): list_name.append(file_path) if file_path[-3:] == ".png" else None else: listdir(file_path, list_name)</pre>	<pre>function withdraw() public returns (uint){ uint amount = balance[msg.sender]; if (amount > 0) { deposits[msg.sender] = 0; msg.sender.transfer(amount); emit withdrawDone(amount, balance[msg. sender], msg.sender); return amount; }</pre>
Ground Truth CODE-NN Code2seq RL-Guided CoCoSum CAST SG-Trans CodeBERT SmartDoc CoSS	<p>save a string array as a file. write array to new table has new line. this method is called when file to file. save array the the the. save a table as a file. write string array table to a file. save array to a file with writer. save array of strings into a file. save a string array to a table file. - save a string array into a file.</p>	<p>Find all png images under the path recursively. append png file name to path. append png files path to the list. list all the png files within the path. list all png images in the folder and add to list. list file path end with png. add png images to the list. list all png files under file path. find png images under the path. - list png images under the path recursively.</p>	<p>Withdraw all deposits of the user. withdraw balance and return to msg.sender. withdraw user balance if done. withdraw uint amount and transfer uint amount to user. withdraw amount balance of the account. send user deposits to the account. withdraw deposits and send to sender. withdraw deposits of the account. withdraw the amount user deposits. withdraw balance of the sender. withdraw all balance of the user account.</p>
Code Snippet			

corresponding summary. On the contrary, CoSS generates the word “recursively”, which is consistent with the word shown in the ground truth. As shown in Fig. 7, the red directed edges in the CFG of case 2 represent loop relationships. For CoSS, the semantic features of these edges are aggregated to neighbor nodes, making them have high attention weights on the generated word “recursively”. This further illustrates how the structural information contained in the CFG improves the quality of the generated summaries.

Table 6 presents the code token and statement attention heatmaps of CoSS for the cases. The marker S_n corresponds to the n -th line of the snippet. We can see that code token attentions focus on semantic-significant tokens such as “png”, “balance”, and method names. However, not all the statements with high attention weights contain such semantic-significant tokens. In case 2, S_5 (`list_name.append(file_path)`) has no individual high-weighted tokens according to the code token attention heatmap, while the generated word “list” pays the highest attention on S_5 . It indicates that code statements that have unique semantics and statement-level attentions are not the simple averages of the in-line token attentions. Thus, it makes sense for CoSS to focus on both code tokens and statements semantics using multi-head attention. Moreover, although as we saw in the heatmaps of case 3, the attention weights of the language-specific words “function” and “public” are very small, which means that these words contribute little to code semantics, they are useful for locating key tokens. For example, “function” is generally followed by a function name in Solidity, and the semantics of function names are vital to code summarization. The lack of these language-specific words makes model difficult to locate such key tokens. Therefore we regard these language-specific words as prevalent and choose to keep them.

In addition, we explored common cases that CoSS failed

to handle properly. It is observed that CoSS cannot handle the source code obtained by decompiling. The compiled program uses meaningless identifiers to replace custom information in the original code, such as variable names, method names, etc., while maintaining the same functionality. CoSS performs poorly on such code snippets because the model cannot capture the semantics of method names, variable names, and the semantics of statements in which variable names are located. This also applies to other code text-based approaches.

6 HUMAN EVALUATION

TABLE 7: The metrics in the questionnaire.

Metric	Explanation
Similarity	The similarity between the generated summaries and references.
Naturalness	The grammaticality and fluency of the generated summaries.
Informativeness	The amount of content carried over from the input code to the generated summaries (ignoring fluency of the text).

Since subjective metrics can not fully reflect human perceptions of the auto-generated summaries, we conducted a human evaluation² among experienced programmers. In the following, we describe the survey design before showing the evaluation results.

6.1 Survey Design

The survey was designed in the form of a questionnaire. We performed stratified sampling from our datasets, evenly split each dataset into ten subgroups based on the code length, and randomly selected five code snippets from

2. The study was approved by the Faculty Human Ethics Advisory Group of Deakin University.

TABLE 8: The results (Standard Deviation σ in Parentheses) of human evaluation (*p < 0.05). Simi.: Similarity, Nat.: Naturalness, Info.: Informativeness.

Approach	Java			Python			Solidity		
	Simi.	Nat.	Info.	Simi.	Nat.	Info.	Simi.	Nat.	Info.
CODE-NN	2.93 (1.16)	3.38 (0.91)	2.20 (1.35)	2.32 (1.34)	3.25 (0.87)	2.11 (1.31)	2.17 (1.15)	2.89 (0.77)	2.10 (1.18)
Code2seq	3.13 (1.40)	3.57 (0.88)	2.87 (1.11)	2.59 (1.28)	3.35 (0.75)	2.49 (1.23)	2.43 (1.05)	3.01 (0.82)	2.56 (1.20)
Hybrid-DRL	3.41 (1.47)	3.90 (0.94)	2.93 (0.96)	2.77 (1.22)	3.54 (0.78)	2.38 (1.17)	2.46 (1.06)	2.87 (0.73)	2.24 (1.13)
RL-Guided	4.09 (1.32)	4.38 (1.09)	3.88 (1.04)	3.97 (1.18)	4.23 (0.75)	3.65 (0.90)	3.44 (0.98)	3.92 (0.65)	3.33 (0.89)
CoCoSum	3.87 (0.94)	4.42 (0.94)	4.03 (0.74)	3.72 (1.07)	4.34 (0.77)	3.81 (1.04)	3.15 (1.12)	4.01 (0.88)	3.45 (1.21)
CAST	3.73 (1.05)	4.33 (1.16)	3.94 (0.91)	3.76 (1.12)	4.18 (0.89)	3.70 (1.20)	3.21 (1.04)	3.84 (0.84)	3.29 (0.95)
SG-Trans	4.17 (1.39)	4.19 (0.89)	4.22 (1.13)	3.45 (1.15)	3.98 (0.85)	3.22 (1.13)	2.95 (0.87)	3.51 (0.92)	2.87 (1.01)
CodeBERT	4.34 (1.03)	4.61 (1.11)	4.35 (1.09)	4.14 (0.98)	4.42 (1.12)	4.06 (1.06)	2.05 (1.14)	2.02 (0.85)	1.87 (1.05)
SmartDoc	-	-	-	-	-	-	2.24 (1.04)	3.29 (0.87)	2.53 (0.92)
CoSS	4.45 (1.15)	4.67 (1.02)	4.42 (1.10)	4.36 (1.04)	4.60 (1.04)	4.24 (0.97)	3.66 (0.95)	4.09 (0.79)	3.57 (1.11)

each subgroup (50 in total). For each selected snippet, we presented the corresponding summaries generated by CoSS and baselines as code comments to build $\langle \text{code}, \text{comment} \rangle$ pairs for evaluation. Following the previous work [43, 47], our survey focuses on three metrics: Similarity, Naturalness, and Informativeness, as Table 7 shows. All the scores are integers, ranging from 1 to 5, following the Likert scale (the higher the better). We invite 30 volunteers who have 3-4 years of developing experience and group them by the programming language they specialize in. Each group has ten volunteers. Volunteers were asked to read the $\langle \text{code}, \text{comment} \rangle$ pairs of their major programming languages, and record the average scores as the overall evaluation results. Volunteers can also provide supplementary explanations for their scores.

6.2 Result Analysis

Table 8 shows the evaluation results. The difference in the standard deviation of the approaches is very small, indicating that their scores are about the same degree of concentration. CoSS is at least 2.5%/1.3%/1.6%(Java), 5.3%/4.0%/4.4%(Python) and 6.3%/2.0%/3.4%(Solidity) better than baselines in Similarity, Naturalness, and Informativeness, respectively. In particular, the Naturalness scores of all approaches remain high, which means almost all generated user summaries are grammatical and fluent. One of the most common issues is that poor naming conventions (meaningless naming, anti-camel-case rule, etc.) of source code can greatly diminish the naturalness of the generated summaries, especially for code-token-only-based approaches. For CoSS, low scores mainly concentrate on complicated summaries with more than 20 words. If a code snippet has a multi-layer nested structure, the corresponding summary is generally long and also gets relatively low scores in all three metrics. All the p-values are smaller than 0.05, proving the improvements in CoSS are statistically significant.

7 THREATS TO VALIDITY

- **Datasets.** Although CoSS is designed as a general DCS approach for different programming languages, we conduct experiments only on Java, Python, and Solidity datasets. Evaluations on more extensive datasets are essential before generalizing CoSS to other programming languages.

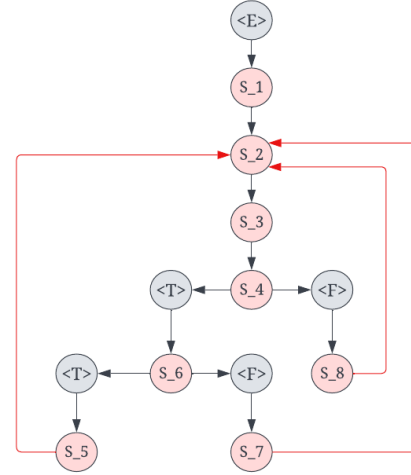


Fig. 7: The CFG of case 1 shown in Table 6.

- **Evaluation Metrics.** We adopt four popular automated metrics for evaluation. However, those metrics are not always fair on different datasets. For example, METEOR is based on the thesaurus of WordNet, while WordNet does not include many blockchain domain-specific terms, which decreases the score of CoSS on the Solidity dataset. Moreover, objective metrics can not completely reflect real human judgments as subjective evaluation does, which means developers do not always benefit from higher-scoring code comments [48] in code comprehension. We will conduct a user study in the future to evaluate the contributions CoSS can provide in real-world development.
- **Model Design.** We adopt original architectures of core components, such as Vanilla Transformer, while ignoring their SOTA variants [49] so as to focus on their core philosophy. In the future, we will explore replacing individual components with such SOTA variants. It is likely to lead to further performance and efficiency promotions.

8 RELATED WORKS

8.1 Rule- and IR-based code summarization

Rule-based approaches mainly depend on software word usage models (SWUMs) or code structure stereotypes to identify the main function of code snippets and generate

corresponding summaries based on templates. SWUM focuses on identifying three elements of codes: action, theme, and possible secondary arguments. For example, if the signature of a method is `list.add(item i)`, SWUM analysis can identify that the action is `add`, the theme is `item`, and the secondary argument is `(to) list`. Hill *et al.* [50] use SWUM to analyze the signatures of Java methods to generate code summaries. On this basis, Sridhara *et al.* [51] propose a heuristic method to generate summaries which can describe the role of the method parameters. McBurney *et al.* [52, 53] collect the method contextual data (method invocation information) and subsequently describe how the methods are used based on the keywords in the context. SWUM is used to identify parts of speech of keywords here. A stereotype is a high-level abstraction of the type and role of a code structure, e.g., class, method, code change, etc. Stereotype information could help to determine the type of code structure that needs to be summarized. For example, if a method aims to instantiate a class into an object, it can be classified as a constructor. A representative stereotype-based approach is proposed by Abid *et al.* [54], which determines the stereotypes of C++ methods based on static analysis and a series of heuristic rules, and then generates code summaries with predefined templates.

For the IR-based approaches, Haiduc *et al.* [6] makes the first attempt to explore such technique for code summarization. Their proposed approach consists of two main steps: extracting words from each function or class for corpus construction and identifying the most relevant words in the corpus. Some approaches collect code-related knowledge from software repositories to generate higher-quality summaries. To name a few, Rahman *et al.* [55] explore the use of crowdsourced knowledge (e.g., post popularity, relevance, comment rating, number of words included, etc.) from StackOverflow to assist in summary generation. In addition, Vassallo *et al.* [56] develop the CODES tool which extracts candidate method comments from StackOverflow posts and generates JavaDoc-based descriptions. Moreover, Wong *et al.* [57] propose an approach named CloCom. It searches for similar code segments from Github through code cloning methods and describes the target code using the comments extracted from searched results.

8.2 Deep learning-based code summarization

Since 2016, the research has concentrated on modeling code summarization as a neural machine translation (NMT) problem. Those approaches employ deep learning techniques to capture complex code features and generate summaries based on encoder-decoder structures. For instance, Iyer *et al.* [7] first investigate such techniques and propose CODENN which is based on LSTM and an attention mechanism in both encoding and decoding. Additionally, Hu *et al.* [9] propose the DeepCom approach to analyze the structural and semantic information of Java methods by means of AST, and subsequently convert AST into sequences. Another example is Code2seq [45], which represents a code snippet as the set of compositional paths in its AST and adopts attention to select relevant paths while decoding. Furthermore, Leclair *et al.* [58] propose the ast-attndgru model, which first considers two types of code modalities:

token-based representation (i.e., code is simply considered as text) and AST-based representation. CAST [13] follows such input modalities and innovatively splits AST into subtrees, since AST is typically deep and hard to handle.

Researchers also explore techniques such as RL and GNN to improve model performance further. For example, Leclair *et al.* [12] adopt the GNN-based encoder from graph2seq to embed the AST of each code snippet. Hybrid-DRL [10] proposed by Wan *et al.* considers both hybrid code representation and deep reinforcement learning. Specifically, it adopts an actor-critic network to solve the exposure bias problem faced in decoding for performance improvement. Next, Wang *et al.* [25] extend Hybrid-DRL to RL-Guided by incorporating a flattened CFG as an additional modality and employing an attention mechanism. Although both RL-Guided and CoSS adopt CFG features, there are three advantages of CoSS over RL-Guided. First, RL-Guided does not deal with the graph structure directly but transforms it into a text-like sequence by traversing the CFG in depth-first order. However, it is generally accepted that a graph cannot be restored from a sequence generated by a classical traversal method, which causes a loss of graph structural information. CoSS uses GAT to embed CFG, which is able to handle graph structures directly, and simultaneously perform information propagation between neighbors as well as parameter updates. Second, RL-Guided adopts code tokens + AST + CFG as the input modalities, while CoSS does not use AST but focuses on obtaining structural information from CFG. This makes the execution efficiency of CoSS much higher than RL-Guided, as experimentally demonstrated. Finally, RL-Guided loses edge features when traversing. For example, branch conditions (True/False) contained in edges are lost in flattened text sequences. CoSS handles this problem by aggregating the features of edges to neighboring nodes during the learning process, to preserve the complete information of the graph.

Apart from AST or token-based approaches, researchers consider other code representations and modalities, such as CPG, DFG, IVFG (interprocedural alias-aware value-flow graph), and UML class diagram Liu *et al.* [44] are the first to use CPG to represent source code in code summarization. On this basis, they combine a retrieval-based augmentation mechanism and a hybrid GNN framework which better captures global graph information than traditional GNN. Gao *et al.* propose SG-Trans [41], a DFG-based approach which extracts global syntactic structure from DFG and injects it into the Transformer. IVFG is built upon LLVM-IR by considering program interprocedural context-sensitive and alias-aware value-flows. Sui *et al.* [59] propose a novel code embedding approach, Flow2vec, to extract deeper code structure information by embedding IVFG in a lower dimension while preserving context-sensitive transitivity. Nevertheless, the IVFG constructor SVF [60] supports only C and C++ programs. Cocosum proposed by Wang *et al.* [46] extracts inter-class context from UML class diagrams and embeds it using MRGNN, then generates code summaries through a two-level attention-based decoder. There are more code representation learning methods that have been applied to other downstream tasks. Wang *et al.* [61] propose a unified framework, Code-MVP, integrating different code views (Code tokens, AST, CFG, and Program Transform-

mations) and learning complementary information among them via contrastive pre-training. Code-MVP demonstrates its superiority on code retrieval, code similarity, and code defect detection tasks over five datasets. Hua *et al.* [62] and Wan *et al.* [63] use a hybrid code representation, preserving code features from code tokens, AST, and CFG in code clone detection code retrieval tasks, respectively. Although these three approaches focus on other code-related tasks, we evaluate the performance of using such hybrid code representations (combinations of Code tokens, AST, and CFG) under the CoSS framework for code summarization in the ablation study, see Section 5.5.

9 CONCLUSION

In this paper, we propose a model named CoSS for automated code summarization. In contrast to existing approaches, we implement a multi-head attention mechanism in both encoder and decoder, making CoSS pay attention to token- and statement-level semantics simultaneously. Since statements can be represented as CFG nodes, we take CFG as one of the modalities and innovatively adopt GAT embedding CFGs to obtain control flow features of the code. Then, we design a Transformer-based joint decoder to generate code summaries. The experimental results show that CoSS achieves SOTA performance and competitive efficiency on Java, Python, and Solidity datasets. We believe that our insight on attention mechanisms, input modalities, and encoding-decoding designs would benefit future studies on code summarization.

ACKNOWLEDGEMENT

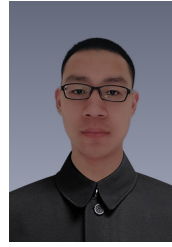
This work was supported in part by the Australian Research Council under grants DP220100983 and LP190100594.

REFERENCES

- [1] N. Nazar, Y. Hu, and H. Jiang, "Summarizing software artifacts: A literature review," *Journal of Computer Science and Technology*, vol. 31, no. 5, pp. 883–909, 2016.
- [2] 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.
- [3] T. Roehm, R. Tiarks, R. Koschke, and W. Maalej, "How do professional developers comprehend software?" in *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 2012, pp. 255–265.
- [4] L. Shi, H. Zhong, T. Xie, and M. Li, "An empirical study on evolution of api documentation," in *International Conference on Fundamental Approaches To Software Engineering*. Springer, 2011, pp. 416–431.
- [5] A. Forward and T. C. Lethbridge, "The relevance of software documentation, tools and technologies: a survey," in *Proceedings of the 2002 ACM symposium on Document engineering*, 2002, pp. 26–33.
- [6] S. Haiduc, J. Aponte, L. Moreno, and A. Marcus, "On the use of automated text summarization techniques for summarizing source code," in *2010 17th Working Conference on Reverse Engineering*. IEEE, 2010, pp. 35–44.
- [7] 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.
- [8] M. Allamanis, E. T. Barr, P. Devanbu, and C. Sutton, "A survey of machine learning for big code and naturalness," *ACM Computing Surveys (CSUR)*, vol. 51, no. 4, pp. 1–37, 2018.
- [9] X. Hu, G. Li, X. Xia, D. Lo, and Z. Jin, "Deep code comment generation," in *Proceedings of the 26th conference on program comprehension*, 2018, pp. 200–210.
- [10] 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.
- [11] M. Allamanis, M. Brockschmidt, and M. Khademi, "Learning to represent programs with graphs," *arXiv preprint arXiv:1711.00740*, 2017.
- [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*, 2020, pp. 184–195.
- [13] E. Shi, Y. Wang, L. Du, H. Zhang, S. Han, D. Zhang, and H. Sun, "Cast: Enhancing code summarization with hierarchical splitting and reconstruction of abstract syntax trees," *arXiv preprint arXiv:2108.12987*, 2021.
- [14] J. Gu, Z. Chen, and M. Monperrus, "Multimodal representation for neural code search," in *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2021, pp. 483–494.
- [15] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang *et al.*, "Codebert: A pre-trained model for programming and natural languages," *arXiv preprint arXiv:2002.08155*, 2020.
- [16] T. D. LaToza, G. Venolia, and R. DeLine, "Maintaining mental models: a study of developer work habits," in *Proceedings of the 28th international conference on Software engineering*, 2006, pp. 492–501.
- [17] J. Singer, T. Lethbridge, N. Vinson, and N. Anquetil, "An examination of software engineering work practices," in *CASCON First Decade High Impact Papers*, 2010, pp. 174–188.
- [18] D. Bruschi, L. Martignoni, and M. Monga, "Detecting self-mutating malware using control-flow graph matching," in *International conference on detection of intrusions and malware, and vulnerability assessment*. Springer, 2006, pp. 129–143.
- [19] S. Cesare and Y. Xiang, "Classification of malware using structured control flow," in *Proceedings of the Eighth Australasian Symposium on Parallel and Distributed Computing-Volume 107*, 2010, pp. 61–70.
- [20] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4–9, 2017, Long Beach, CA, USA*, 2017, pp. 5998–6008.
- [21] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Lio, and Y. Bengio, "Graph attention networks,"

- arXiv preprint arXiv:1710.10903*, 2017.
- [22] R. Rosenfeld, "Two decades of statistical language modeling: Where do we go from here?" *Proceedings of the IEEE*, vol. 88, no. 8, pp. 1270–1278, 2000.
 - [23] S. Wang, D. Chollak, D. Movshovitz-Attias, and L. Tan, "Bugram: bug detection with n-gram language models," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, 2016, pp. 708–719.
 - [24] E. Grave, A. Joulin, and N. Usunier, "Improving neural language models with a continuous cache," *arXiv preprint arXiv:1612.04426*, 2016.
 - [25] 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.
 - [26] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.
 - [27] Z. Yang, Z. Dai, Y. Yang, J. Carbonell, R. R. Salakhutdinov, and Q. V. Le, "Xlnet: Generalized autoregressive pretraining for language understanding," *Advances in neural information processing systems*, vol. 32, 2019.
 - [28] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," *arXiv preprint arXiv:1609.02907*, 2016.
 - [29] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," in *1st International Conference on Learning Representations, ICLR 2013, Scottsdale, Arizona, USA, May 2-4, 2013, Workshop Track Proceedings*, 2013. [Online]. Available: <http://arxiv.org/abs/1301.3781>
 - [30] D.-A. Clevert, T. Unterthiner, and S. Hochreiter, "Fast and accurate deep network learning by exponential linear units (elus)," *arXiv preprint arXiv:1511.07289*, 2015.
 - [31] R. J. Williams and D. Zipser, "A learning algorithm for continually running fully recurrent neural networks," *Neural computation*, vol. 1, no. 2, pp. 270–280, 1989.
 - [32] H. Husain, H.-H. Wu, T. Gazit, M. Allamanis, and M. Brockschmidt, "CodeSearchNet challenge: Evaluating the state of semantic code search," *arXiv preprint arXiv:1909.09436*, 2019.
 - [33] A. V. M. Barone and R. Sennrich, "A parallel corpus of python functions and documentation strings for automated code documentation and code generation," *arXiv preprint arXiv:1707.02275*, 2017.
 - [34] W. Wang, Y. Zhang, Z. Zeng, and G. Xu, "Trans³: A transformer-based framework for unifying code summarization and code search," *arXiv preprint arXiv:2003.03238*, 2020.
 - [35] C. Shi, Y. Xiang, J. Yu, and L. Gao, "Semantic code search for smart contracts," *arXiv preprint arXiv:2111.14139*, 2021.
 - [36] J. Feist, G. Grieco, and A. Groce, "Slither: a static analysis framework for smart contracts," in *2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*. IEEE, 2019, pp. 8–15.
 - [37] 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.
 - [38] C.-Y. Lin, "Rouge: A package for automatic evaluation of summaries," in *Text summarization branches out*, 2004, pp. 74–81.
 - [39] S. Banerjee and A. Lavie, "Meteor: An automatic metric for mt evaluation with improved correlation with human judgments," in *Proceedings of the acl workshop on intrinsic and extrinsic evaluation measures for machine translation and/or summarization*, 2005, pp. 65–72.
 - [40] R. Vedantam, C. Lawrence Zitnick, and D. Parikh, "Cider: Consensus-based image description evaluation," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2015, pp. 4566–4575.
 - [41] S. Gao, C. Gao, Y. He, J. Zeng, L. Y. Nie, and X. Xia, "Code structure guided transformer for source code summarization," *arXiv preprint arXiv:2104.09340*, 2021.
 - [42] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck, "Modeling and discovering vulnerabilities with code property graphs," in *2014 IEEE Symposium on Security and Privacy*. IEEE, 2014, pp. 590–604.
 - [43] X. Hu, Z. Gao, X. Xia, D. Lo, and X. Yang, "Automating user notice generation for smart contract functions," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2021, pp. 5–17.
 - [44] S. Liu, Y. Chen, X. Xie, J. K. Siow, and Y. Liu, "Retrieval-augmented generation for code summarization via hybrid GNN," in *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net, 2021. [Online]. Available: <https://openreview.net/forum?id=zv-typ1gPxA>
 - [45] U. Alon, S. Brody, O. Levy, and E. Yahav, "code2seq: Generating sequences from structured representations of code," *arXiv preprint arXiv:1808.01400*, 2018.
 - [46] Y. Wang, E. Shi, L. Du, X. Yang, Y. Hu, S. Han, H. Zhang, and D. Zhang, "Cocosum: Contextual code summarization with multi-relational graph neural network," *arXiv preprint arXiv:2107.01933*, 2021.
 - [47] B. Wei, Y. Li, G. Li, X. Xia, and Z. Jin, "Retrieve and refine: exemplar-based neural comment generation," in *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2020, pp. 349–360.
 - [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.
 - [49] T. Lin, Y. Wang, X. Liu, and X. Qiu, "A survey of transformers," *arXiv preprint arXiv:2106.04554*, 2021.
 - [50] E. Hill, L. Pollock, and K. Vijay-Shanker, "Automatically capturing source code context of nl-queries for software maintenance and reuse," in *2009 IEEE 31st International Conference on Software Engineering*. IEEE, 2009, pp. 232–242.
 - [51] G. Sridhara, L. Pollock, and K. Vijay-Shanker, "Generating parameter comments and integrating with method summaries," in *2011 IEEE 19th International Conference on Program Comprehension*. IEEE, 2011, pp. 71–80.

- [52] P. W. McBurney and C. McMillan, "Automatic documentation generation via source code summarization of method context," in *Proceedings of the 22nd International Conference on Program Comprehension*, 2014, pp. 279–290.
- [53] —, "Automatic source code summarization of context for java methods," *IEEE Transactions on Software Engineering*, vol. 42, no. 2, pp. 103–119, 2015.
- [54] N. J. Abid, N. Dragan, M. L. Collard, and J. I. Maletic, "Using stereotypes in the automatic generation of natural language summaries for c++ methods," in *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2015, pp. 561–565.
- [55] M. M. Rahman, C. K. Roy, and I. Keivanloo, "Recommending insightful comments for source code using crowdsourced knowledge," in *2015 IEEE 15th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 2015, pp. 81–90.
- [56] C. Vassallo, S. Panichella, M. Di Penta, and G. Canfora, "Codes: Mining source code descriptions from developers discussions," in *Proceedings of the 22nd International Conference on Program Comprehension*, 2014, pp. 106–109.
- [57] 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.
- [58] 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.
- [59] Y. Sui, X. Cheng, G. Zhang, and H. Wang, "Flow2vec: Value-flow-based precise code embedding," *Proceedings of the ACM on Programming Languages*, vol. 4, no. OOPSLA, pp. 1–27, 2020.
- [60] Y. Sui and J. Xue, "Svf: interprocedural static value-flow analysis in llvm," in *Proceedings of the 25th international conference on compiler construction*, 2016, pp. 265–266.
- [61] X. Wang, Y. Wang, Y. Wan, J. Wang, P. Zhou, L. Li, H. Wu, and J. Liu, "Code-mvp: Learning to represent source code from multiple views with contrastive pre-training," *arXiv preprint arXiv:2205.02029*, 2022.
- [62] W. Hua, Y. Sui, Y. Wan, G. Liu, and G. Xu, "Fcca: Hybrid code representation for functional clone detection using attention networks," *IEEE Transactions on Reliability*, vol. 70, no. 1, pp. 304–318, 2020.
- [63] Y. Wan, J. Shu, Y. Sui, G. Xu, Z. Zhao, J. Wu, and P. Yu, "Multi-modal attention network learning for semantic source code retrieval," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 13–25.



Chaochen Shi got a B.E. from Beijing University of Posts and Telecommunications, and a M.S. from University College London. He is doing his PhD now as a member of DBIL (Deakin Blockchain Innovation Lab). His research interests are in the area of software engineering, code analysis, and blockchain technology.



Borui Cai received the bachelors degree and masters degree of engineering from the Beihang University (BUAA) in 2010 and 2013, respectively, and the PhD degree from Deakin University in 2021. He is currently a postdoctoral research fellow at School of Information, Deakin University. He worked for Chinese Academy of Sciences between 2013 and 2016, and worked in Xilinx, Inc. in 2017. His research interests include natural language processing, time series analysis, and privacy protection.



Yao Zhao is currently pursuing the Ph.D. degree at the School of Information Technology, Deakin University. She has completed a M.S. degree of Computer Science and Technology at Nanjing University of Aeronautics and Astronautics in 2019. Her research direction includes blockchain application, consensus mechanism, IoT, and edge computing.



analysis and privacy protection.

Longxiang Gao (SM'17) received his PhD in Computer Science from Deakin University, Australia. He is currently a Professor at Qilu University of Technology (Shandong Academy of Sciences) and Shandong Computer Science Center (National Supercomputer Center in Jinan). He was a Senior Lecturer at School of Information Technology, Deakin University and a post-doctoral research fellow at IBM Research & Development, Australia. His research interests include Fog/Edge computing, Blockchain, data



Keshav Sood received his PhD from Deakin University in 2018. Following his PhD, he worked as Research Fellow with The University of Newcastle, NSW, Australia. He worked on the project funded by Defence Science and Technology Group. He is currently a Lecturer in Deakin University, Melbourne. Some of his work is funded by Department of Defense, Australia and Cyber Security Cooperative Research Centre (CSCRC), Australia.



editor of IEEE Signal Processing Letters, the Associate Editor of IEEE Communications Surveys and Tutorials, and the Associate Editor of Computer Standards and Interfaces.

Yong Xiang (SM'12) received the Ph.D. degree in Electrical and Electronic Engineering from The University of Melbourne, Australia. He is a Professor at the School of Information Technology, Deakin University, Australia. His research interests include information security and privacy, data analytics and machine learning, Internet of Things, and blockchain. He has published 7 monographs, over 220 refereed journal articles, and over 100 conference papers in these areas. Professor Xiang is the Senior Area Editor of IEEE Signal Processing Letters, the Associate Editor of IEEE Communications Surveys and Tutorials, and the Associate Editor of Computer Standards and Interfaces.