Automating Comment Generation for Smart Contract from Bytecode

JIANHANG XIANG, The State Key Laboratory of Blockchain and Data Security, Zhejiang University, China
ZHIPENG GAO*, The State Key Laboratory of Blockchain and Data Security, Zhejiang University, China
LINGFENG BAO, The State Key Laboratory of Blockchain and Data Security, Zhejiang University, China and Hangzhou High-Tech Zone (Binjiang) Blockchain and Data Security Research Institute, China
XING HU, The State Key Laboratory of Blockchain and Data Security, Zhejiang University, China
JIAYUAN CHEN, The Ohio State Univeristy, United States
XIN XIA, The State Key Laboratory of Blockchain and Data Security, Zhejiang University, China

Recently, smart contracts have played a vital role in automatic financial and business transactions. To help end users without programming background to better understand the logic of smart contracts, previous studies have proposed models for automatically translating smart contract source code into their corresponding code summaries. However, in practice, only 13% of smart contracts deployed on the Ethereum blockchain are associated with source code. The practical usage of these existing tools is significantly restricted. Considering that bytecode is always necessary when deploying smart contracts, in this paper, we first introduce the task of automatically generating smart contract code summaries from bytecode. We propose a novel approach, named **SmartBT** (**Smart** contract **B**ytecode **T**ranslator) for automatically translating smart contract bytecode into fine-grained natural language description directly. Two key challenges are posed for this task: structural code logic hidden in bytecode and the huge semantic gap between bytecode and natural language descriptions. To address the first challenge, we transform bytecode into CFG (Control-Flow Graph) to learn code structural and logic details. Regarding the second challenge, we introduce an information retrieval component to fetch similar comments for filling the semantic gap. Then the structural input and semantic input are used to build an attentional sequence-to-sequence neural network model. The copy mechanism is employed to copy rare words directly from similar comments and the coverage mechanism is employed to eliminate repetitive outputs. The automatic evaluation results show that SmartBT outperforms a set of baselines by a large margin, and the human evaluation results show the effectiveness and potential of SmartBT in producing meaningful and accurate comments for smart contract code from bytecode directly.

CCS Concepts: • **Software and its engineering → Software creation and management**; **Documentation**.

Additional Key Words and Phrases: Smart Contract, Bytecode, Automatic Comment Generation

*This is the corresponding author

Authors' addresses: Jianhang Xiang, The State Key Laboratory of Blockchain and Data Security, Zhejiang University, Hangzhou,, China; Zhipeng Gao, The State Key Laboratory of Blockchain and Data Security, Zhejiang University, Shanghai,, China; Lingfeng Bao, The State Key Laboratory of Blockchain and Data Security, Zhejiang University, Hangzhou,, China and Hangzhou High-Tech Zone (Binjiang) Blockchain and Data Security Research Institute, Hangzhou,, China; Xing Hu, The State Key Laboratory of Blockchain and Data Security, Zhejiang University, Hangzhou,, China; Jiayuan Chen, The Ohio State Univeristy, Columbus, United States; Xin Xia, The State Key Laboratory of Blockchain and Data Security, Zhejiang University, Hangzhou,, China.

ACM 1557-7392/2024/10-ART
https://doi.org/10.1145/3699597

1 INTRODUCTION

Smart contract, which was created by Nick Szabo [55], is a program (mainly written by Solidity) that can be triggered and executed on the Ethereum Blockchain. It allows developers to write self-executing smart contracts that can be deployed on Ethereum, enabling the development of decentralized applications. The Ethereum blockchain has experienced remarkable growth along the blockchain technology, the average amount of smart contracts deployed on Ethereum has exceeded 4,200 each month, and the average number of active Ethereum addresses has exceeded 82 million per month. As a result, Ethereum has become one of the largest cryptocurrency platforms (with an overall market capitalization surpassing 228.38 billion in USD) and smart contracts are increasingly used to automate financial and business transactions.

Despite the great success of smart contracts, notable concerns have also emerged, especially for novice users who are unfamiliar with smart contracts. Among them, **false advertising** stands out as a crucial issue faced by smart contract end users. For instance, smart contract end users may experience ICO (Initial Coin Offer) scams when investing in cryptocurrencies. The scammers make false claims about the project's innovative technology and/or business logic, enticing investors with promises of substantial returns on investment. However, the smart contracts they deployed lack such actual utility, value, or even existence. The mismatch between the smart contract code implementations and the content described in their white papers, websites, and announcements has led to significant financial losses for investors. As pointed out by [1], ten of the most high-profile ICO scams swindled 687.4 million USD from unsuspecting investors. To help end users find the inconsistency between the smart contract source code and their documentation, researchers have investigated the ways of automatically translating Solidity source code into fine-grained English descriptions [29, 51, 68]. For example, Hu et al. [29] introduced SmartDoc, a deep-learning-based model to generate user notice for smart contracts. In this way, end users without programming knowledge can understand and learn the logic of original smart contracts. Shi et al. [51] presented a reinforcement learning model, named SolcTrans, to generate comments of the Solidity source code via AST traversal and Probalistic Context-Free Grammar rules. **All of the previous studies focus on generating high-level summaries from Solidity source code, however, it is not guaranteed that the Solidity source code of every smart contract can be obtained.**

Typically, when developers deploy smart contracts on the Ethereum blockchain, they first compile the contract source code to bytecode and then the bytecode is deployed to the blockchain. Developers can choose to upload their source code for verification but this is optional. According to statistics obtained by Zellic [4], **they collected more than 30 million smart contracts deployed on the Ethereum blockchain, only 13% of them are associated with source code.** In other words, the existing methods which rely on the availability of Solidity source code, are incapable of handling the majority of smart contracts on the Ethereum blockchain where source code are missing. As a result, the practical usage of these tools is significantly limited and severely restricted.

Although the source code of the smart contracts may not be publicly accessible, the bytecode of every smart contract is available upon deployment on the blockchain. **In this study, we first investigate the possibility of generating smart contract code summaries directly from bytecode, bypassing the existing methods that rely on the original source code.** However, generating comments from bytecode is a difficult task with respect to the following challenges:

(1) *Learning structural information from bytecode.* The Solidity bytecode is generated by compiling Solidity source code into an instruction set that can be read and executed by the Ethereum Virtual Machine (EVM). Unlike human-readable source code, the bytecode is a series of hexadecimal numbers, which are not easily understandable by individuals without specialized knowledge. Therefore, how to extract the information about the contract's structure and behavior as well as learning the logic and functionality pattern encoded within Solidity bytecode presents a significant challenge.

(2) *Semantic gap between bytecode and comments.* The semantic gap refers to the disparity between the low-level EVM bytecode and the high-level explanations conveyed by human-written comments. Compared with source code, the semantic gap between bytecode and natural language comment is even more larger, how to properly fill the gap and effectively transform bytecode into meaningful natural language comments is another challenge for this study.

In this work, to help end users better understand the logic details of the smart contracts deployed on the Ethereum blockchain, we propose a novel neural network model, named SmartBT (Smart contract Bytecode Translator), which can automatically translate smart contract bytecode into human-readable code summaries. The generated code descriptions can be used as function code comments to help users understand and participate in interacting with smart contracts more easily and safely. In particular, we first collect 30,742 $\langle bytecode, comment \rangle$ pairs from 54,739 deployed smart contracts. To extract structural information from the bytecode, we convert each smart contract bytecode into a Control-Flow Graph (CFG) to learn the contract structure from the encoded bytecode. To fill the semantic gap between bytecode and comment, we introduce the IR (Information-retrieval) augmented module to fetch relevant comments from contracts with similar CFGs. Finally, we build an IR-augmented sequence-to-sequence model by incorporating the structural input (i.e., CFGs) and semantic input (i.e., relevant comments). Moreover, we have introduced the *copy* mechanism to copy rare words from the IR module and the *coverage* mechanism to eliminate the word repetition problem. The automatic and human evaluation results show the advantage and superiority of SmartBT for generating comments from bytecode for smart contracts. The paper makes the following contributions:

- The existing comment generation methods rely on the availability of source code, while 90% deployed smart contracts' source code are missing on the Ethereum blockchain. In this study, we first proposed the new task of generating smart contract comments from the bytecode level.
- We build the first dataset for the smart contract bytecode comment generation task. In particular, we have collected 30,742 $\langle bytecode, comment \rangle$ pairs of different functions from 54,739 verified smart contracts. Each function bytecode is converted into an intermediate representation of CFG to provide useful structural information for downstream tasks.
- To the best of our knowledge, SmartBT is the first model to investigate the possibility of generating smart contract comments directly from the bytecode. We introduce the CFG and IR augmented components to fill the gap between the bytecode and natural language comments. We extensively evaluate the SmartBT on real-world deployed smart contracts, SmartBT is shown to outperform several baselines and reduce the user's efforts in understanding smart contracts.
- We have released a replication package [8], including the dataset and source code of SmartBT, to facilitate other researchers and practitioners to repeat our work and verify their own ideas.

The rest of the paper is organized as follows. Section 2 presents the background of our research. Section 4.1 presents our approach details. Section 5 presents the experiment setup. Section 6 presents our research questions and experimental results. Section 7 presents the threats to validity. Section 9 presents related works. Finally, Section 10 presents the conclusion and future work.

2 BACKGROUND

2.1 Smart Contracts

Smart contract, a term coined by Nick Szabo in 1994 [54], was proposed as a computerized transaction protocol that executes the contractual terms of an agreement. Recently, along with development of the blockchain technology, smart contracts can be essentially implemented on top of blockchains (i.e., Ethereum), which are referred to as code scripts to execute certain tasks once predefined conditions are met.

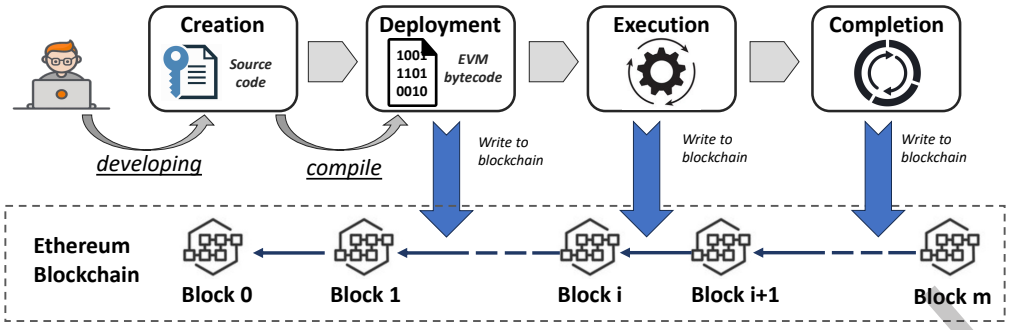


Fig. 1. Life Cycle of Smart Contracts

Particularly, smart contracts are programs that run on the Ethereum blockchain. The contractual clauses are converted into executable computer programs. The logical connections between contractual clauses are also been preserved in the form of logical flows in programs (e.g., the `if-else` statement). If any condition in a smart contract is satisfied, the triggered statement will automatically execute the corresponding function in a predictable manner. The execution of each contract statement is then recorded as an immutable transaction stored in the blockchain system. For example, Bob and Alice have an agreement on the penalty of violating the contract. If Bob breaches the contract, the corresponding penalty (as specified in the contract) will be automatically deducted from Bob's deposit.

As shown in Fig. 1, the life cycle of smart contracts consists of four consecutive phases: 1) *Smart Contract Creation:* Several involved parties (e.g., stakeholders, lawyers) will first reach a contractual agreement after discussions. Then software developers convert this agreement written in natural languages into smart contract(s) written in computer languages (e.g., Solidity, Vyper). 2) *Smart Contract Deployment:* The validated smart contracts then can be deployed on top of the Ethereum blockchain. Since Ethereum uses EVM (Ethereum Virtual Machine) to execute smart contracts, to deploy a smart contract on Ethereum, the contract source code (e.g., Solidity) needs to be compiled into EVM bytecode and the bytecode will be stored on the blockchain. 3) *Smart Contract Execution:* After deployment, the smart contracts are triggered by events. These events can be external (e.g., payment received) or internal (e.g., specific date or time). Once an event satisfies the predefined conditions, the corresponding statements will be automatically executed. A transaction is executed and validated by miners in the Ethereum blockchain. 4) *Smart Contract Completion:* After execution, the transactions during the execution, as well as the updated states, are permanently stored in blockchains. Meanwhile, the digital assets have been transferred from one party to another (e.g., money transfer from the buyer to the supplier). Notably, only EVM bytecode is deployed and stored on the blockchain, while the source code remains off-chain and inaccessible.

From the developers' perspective, the developer first needs to write source code for smart contracts with programming languages (e.g., Solidity). Then the developer uses compilers (e.g., `Solc`) to convert source code into EVM bytecode. The developer deploys the bytecode to the Ethereum blockchain and saves it at an address. Particularly, two types of EVM bytecode are associated with deployment, i.e., *creation code* and *runtime code*. *Creation code* is responsible for the creation of the contract (e.g., setting up the constructor and initializing constructor variables), which is executed only once during deployment. Unlike *creation code*, *runtime code* doesn't include the constructor details. Once the *creation code* is executed, *runtime code* is stored on-chain for anyone to interact with. *Runtime code* describes the contract, any on-chain interaction with the smart contract means an interaction with the *runtime code*. In this work, we refer to smart contract *runtime code* as bytecode for short in the rest of this paper.

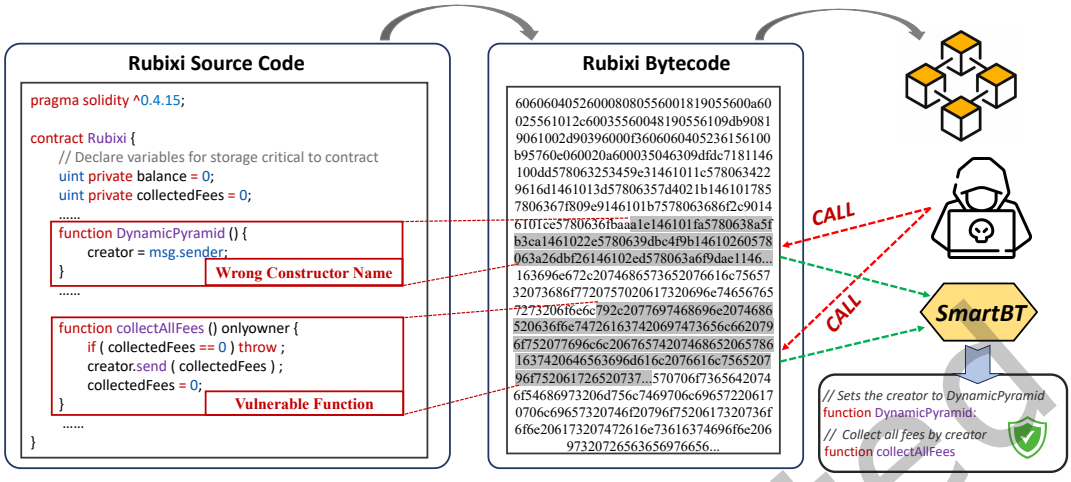


Fig. 2. Motivating Example of Using SmartBT

2.2 Neural Machine Translation

Neural Machine Translation (NMT) is an end-to-end learning framework for automated translation. It is a deep learning-based approach and has made rapid progress in recent years [18, 20, 29, 30]. NMT has shown impressive results surpassing those of phrase-based or rule-based systems while addressing shortcomings such as the need for manually crafted features. NMT models usually consist of an encoder-decoder structure. The encoder encodes the input sequence into a fixed-length vector, which represents the semantic and contextual information of the source language sentence. The decoder gradually generates translated output sequences based on the vectors encoded by the encoder and the previously generated target language parts.

NMT has proven to be effective in bridging the gap between different languages in natural language processing. The NMT framework has also been successfully applied to various software engineering tasks [18, 19, 29, 44], including comment generation [29]. Software engineering researchers view comment generation as a variant of the machine translation problem between source code and natural language. However, all previous studies focus on generating comments from source code. In this study, we first explore the possibility of whether the NMT framework can be applied to comment generation from Solidity bytecode.

3 MOTIVATION

In this section, we provide a motivating example to explain the usage of our tool in practice. Fig. 2 shows a common vulnerability, namely *wrong constructor name*, in smart contracts. Constructors are special functions that are called only once during the contract creation. They often perform critical, privileged actions such as setting the owner of the contract. Constructor names have to be the same as the contract class that contained it. If the constructor method is inconsistent with the contract class (a.k.a., missnamed), security issues will be introduced.

Fig. 2 demonstrates a real smart contract with the *wrong constructor name* weakness. The smart contract `Rubixi` uses `DynamicPyramid` instead of `Rubixi` as a constructor. Because of this inconsistency, the contract did not assign the owner upon contract creation. As a result, anyone who calls this `DynamicPyramid()` function can assign themselves as the owner of the contract. After granting the ownership, then they can easily collect contract fees generated by participating players (e.g., calling function `collectAllFees()`). With our tool, even if contract `Rubixi` is not open-sourced, SmartBT can successfully generate a comment, i.e., *sets the creator to*

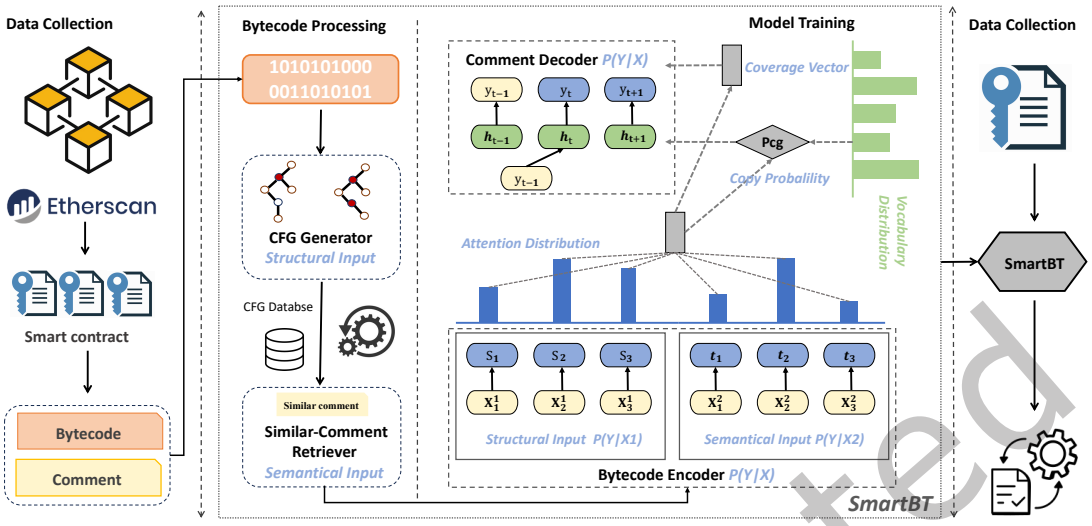


Fig. 3. Workflow of SmartBT

DynamicPyramid, for the `DynamicPyramid()` function from its bytecode. It can help users notice this inconsistency between contract name (`Rubixi`) and constructor name (`DynamicPyramid`), and better make informed decisions.

4 APPROACH

In this section, we first define the task of comment generation from bytecode, then present the details of our approach. Figure 3 demonstrates the workflow of the SmartBT. Our approach is primarily composed of three stages: bytecode preprocessing, model training, and comment generation.

4.1 Task Definition

The main purpose of our work is to improve the understanding of a smart contract from its bytecode directly. We thus propose a novel task in this paper - generating smart contract code comments from their corresponding bytecode. Inspired by the great success of using the NMT framework in previous studies [29], We formulate our task as a sequence-to-sequence learning problem.

Given that X is the input bytecode sequence of a smart contract function, our target is to generate its corresponding comment Y describing the function. In particular, our objective is to learn the underlying conditional probability distribution $P_\theta(Y|X)$. In other words, the goal is to train a model θ using $\langle X, Y\rangle$ pairs such that the probability $P_\theta(Y|X)$ is maximized over the given training dataset. The training objective function can be formulated as maximizing the log-likelihood:

$$\theta^* = \arg\max_\theta \sum_{\langle X,Y\rangle} \log P_\theta(Y|X). \tag{1}$$

4.2 Bytecode Processing

Because we directly deal with smart contracts bytecode, two key challenges are posed for this study.

- **Challenge 1:** How can we effectively capture the structural information and logic details encoded in smart contract bytecode?

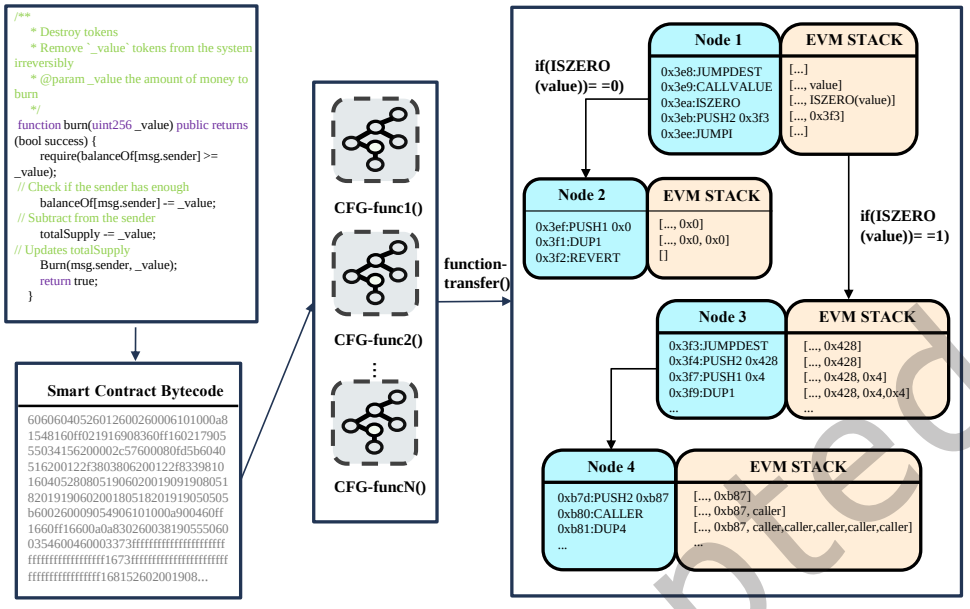


Fig. 4. Example of Solidity Source Code, Bytecode and CFG

- **Challenge 2:** How can we effectively bridge the gap between semantic gap between the bytecode and natural language comments?

To address these two key challenges, we process the smart contract bytecode with two components: a **CFG Generator** and a **Similar-Comment Retriever**. **CFG Generator** is responsible for constructing input containing structural information, **Similar-Comment Retriever** is responsible for constructing input associated with semantic information, then these two types of inputs are used to train an end-to-end model for automatically generating comments from contract bytecode.

4.2.1 ***CFG Generator.*** To deploy a smart contract to Ethereum, its source code needs to be compiled to bytecode and stored on the blockchain. There are 140 unique opcodes by April 2019 [65], and each opcode is represented by a hexadecimal number. EVM (Ethereum Virtual Machine) uses these opcodes to execute the task. When a transaction needs to be executed, EVM will first split the bytecode into bytes. Each byte represents a unique instruction called opcode. For example, for the bytecode `0x6070604001`, EVM first splits the bytecode into bytes (i.e., `0x60`, `0x70`, `0x60`, `0x40`, `0x01`). EVM then executes the first byte `0x60` which refers to the opcode `PUSH1`. `PUSH1` pushes the one byte data (`0x70`) to the EVM stack. Then EVM executes the third byte `0x60` and pushes `0x40` into the stack. Finally, EVM executes `0x01` which refers to opcode `ADD`. `ADD` obtains two values from the stack (i.e., `0x70` and `0x40`) and perform the sum operation. As can be seen, the smart contract bytecode contains detailed contract execution logic and structural information. To capture the structural information and detailed logic, we convert the smart contract bytecode into CFG (Control-Flow Graph) representations. A CFG is a graphical representation of the code control flow during the execution of a contract. Each node in the CFG represents a basic block, which contains a straight-line piece of opcode without any jumps or jump targets. Blocks are connected by edges which represent jumps to form a control flow graph. The CFG representations can capture all the possible flows of execution of all code blocks and can reflect the real-time execution of a code fragment, which is valuable for extracting structural information encoded within the contract bytecode.

For a given smart contract bytecode, we first extract the CFG for each function within the smart contract. As shown in Figure 4, each smart contract may contain a number of functions, after the extraction process, each function is converted into a CFG. In this work, we use the tool `evm_cfg_builder` for CFG extraction, the tool can reliably recover a CFG from EVM bytecode using a dedicated value set analysis. However, the CFG cannot be directly fed into a sequence model, we further traverse the CFG by utilizing DFS (Depth-First-Search) and generate the target CFG sequence. In particular, we first identify the root node of each CFG, we then serialize CFG into a sequence by using DFS algorithms. The symbol `->` is used when traversing from one block to another, maintaining the structural information of different blocks. So far, we have converted each function of the bytecode into a CFG representation by our **CFG Generator**, and the structural inputs are prepared for the downstream model training.

4.2.2 ***Similar-comment Retriever***. Compared with source code, the semantic gap between bytecode and natural language code descriptions is even larger, which poses another key challenge for translating bytecode into code summaries. To bridge the semantic gap between bytecode and comments, we introduced another component, namely **Similar-comment Retriever**. This component is responsible for retrieving similar comments from our codebase and using them as semantic inputs for our model. Our goal is to leverage these retrieved similar comments as valuable references for improving our translation process.

Specifically, we have stored all the functions' CFG and their corresponding comments as our codebase. When presented with the bytecode of a new function, we initially convert it into its CFG using the **CFG Generator**. Subsequently, we employ an information retrieval algorithm, specifically BM25 [46] for this study, to query our codebase and search for similar functions by comparing similarities between CFGs. The top similar functions are returned and their corresponding comments are extracted as our semantic inputs. The **Similar-comment Retriever** component acts as a critical link in our model, providing a way to connect bytecode with semantic relevant natural language descriptions, which can enhance the final inference of our approach when generating code summaries. So far, the semantic inputs have also been prepared for our model building.

4.3 Bytecode Encoder

After the structural inputs, i.e., CFGs, and semantic inputs, i.e., similar-comments, are prepared, we concatenate these two inputs and feed them sequentially into our Bytecode Encoder. We add a special token `[SEP]` between semantic input and structural input to further separate natural language (i.e., comments) and code implementations (i.e., CFGs), which has been proven to be effective for bridging the gap between heterogeneous data [24, 61]. The Bytecode Encoder uses Bidirectional Recurrent Neural Networks (BRNN), which can improve the model's ability to understand the input sequence from both forward and backward directions, capture more comprehensive context information, and help the decoder generate more accurate target sequences. The Bytecode Encoder aims to learn contextual representations from bytecode processing outputs (including generated CFGs and retrieved similar-comments). The concatenated inputs, including the smart contract CFG and its associated similar comments, are embedded into a vector before being fed into the Bytecode Encoder. The calculation equation of forward RNN and reverse RNN are:

$$h_t^f = \text{BRNN}^f\left(x_t, h_{t-1}^f\right), \tag{2}$$

$$h_t^b = \text{BRNN}^b\left(x_t, h_{t+1}^b\right). \tag{3}$$

Here h_t^f represents the hidden state of forward RNN at time step t, h_t^b represents the hidden state of reverse RNN at time step t, and x_t represents the input sequence at time step t. The hidden states of forward RNN and reverse RNN can be calculated by the order of time steps, or by reverse calculation (starting from the last time step). In BRNN, the hidden states of these two directions can be concatenated or merged to obtain a complete bidirectional

representation. Here, we mainly take the concatenation operation. As shown in Eq. (4), h_t^f is the forward hidden state, h_t^b is the reverse hidden state, and h_t is the hidden state after splicing. Here $[;]$ represents the splicing operation. By concatenation, BRNN provides richer bidirectional context information for subsequent decoding.

$$h_t = \left[h_t^f; h_t^b\right]. \tag{4}$$

4.4 Comment Decoder

SmartBT uses a 2-layer LSTM network as its decoder, which refers to a specific recurrent neural network structure that consists of two LSTM layers stacked together. Each LSTM layer has its own set of LSTM cells responsible for capturing and storing information over time. The output of one LSTM layer serves as the input to the next layer, enabling the network to learn hierarchical representations of sequential data. This deeper architecture enables the model to capture more complex dependencies and make more complex predictions than single-layer LSTM. For the decoding process, we first have the following assumptions: t is the time step, and h_t is the hidden states at t time step. In our task, each token of the contract function comment will be embedded into a vector, and we assume that $Cword_t$ is the target word at t of the ground truth comments, y_t is the embedding vector of $Cword_t$. For the first and second layers of LSTM, the calculation process is as follows:

$$L1_t = \text{LSTM}_1\left(y_t, h_{t-1}\right), \tag{5}$$

$$L2_t = \text{LSTM}_2\left(y_t, \mathbf{L1}_t\right). \tag{6}$$

During training, at each time step t, the first layer LSTM takes the embedding vector y_t of the target word $Cword_t$ and the previous state h_{t-1} as input, and concatenates them to produce the output hidden state of the first layer. For the second layer of LSTM, the initial value of h_t is the output hidden state of the first layer of LSTM. The decoder produces one symbol at a time and stops when the `END` symbol is emitted. The only change with the decoder at the testing time is that it uses the output from the previous word emitted by the decoder in place of $Cword_{t-1}$, since there is no access to a ground truth then.

4.5 Incorporating Attention Mechanism

In a traditional NMT framework, the encoder processes the input sequence and encodes it into a fixed-size context vector. This fixed-size context vector can become a bottleneck when dealing with long sequences or capturing important information from different parts of the input. By using Bahdanau Attention [10] as the global attention mechanism, our **Comment Decoder** can focus on different parts of the input sequence dynamically. In particular, We model the attention [10] distribution over words in the source input sequence. We calculate the attention (a_i^t) over the i^{th} input sequence token as:

$$e_i^t = v^t \tanh\left(W_{eh} h_i + W_{sh} s_t + b_{att}\right) \tag{7}$$

$$a_i^t = \text{softmax}\left(e_i^t\right) \tag{8}$$

Here, v^t, W_{sh} and b_{att} are model parameters to be learned, and h_i is the concatenation of forward and backward hidden states of our **Bytecode Encoder**. We use this attention a_i^t to generate the context vector c_t^* as the weighted sum of encoder hidden states :

$$\mathbf{c}_t^* = \sum_{i=1,..,|\mathbf{x}|} a_i^t \mathbf{h}_i \tag{9}$$

We further use the c_t^* vector to obtain a probability distribution over the words in the vocabulary as follows:

$$P = \text{softmax}\left(\mathbf{W}_v[s_t, c_t^*] + b_v\right) \tag{10}$$

where W_v and b_v are model parameters. Thus during decoding, the probability of generating a target word is $P(Cword)$. During the training process for each word at each timestamp, the loss associated with the generated comment is:

$$Loss = -\frac{1}{T}\sum_{t=0}^{T} logP(Cword_t) \tag{11}$$

The *attention* mechanism allows the model to focus on the most relevant parts of the input sequence as needed. Instead of relying solely on the last hidden state, the *attention* mechanism allows our **Comment Decoder** to consider all the hidden states from the **Bytecode Encoder**. It assigns different attention weights to each hidden state, indicating its relevance to the current decoding step. This ability to amplify the signal from the CFG input and similar comments input makes attention models produce better results than models without attention.

4.6 Incorporating Copy Mechanism

The *copy* mechanism [23] is commonly used in tasks such as machine translation and text summarization. It is used to facilitate the model to copy tokens from the input sequence to the generated output sequence. This is because some words are much less frequent than others, thus it is highly unlikely for a decoder that is solely based on a language model to generate such a word with very rare occurrences in a corpus. In such cases, the possibly rare words in the input sequence might be required to be *copied* from our input sequence to the target comment. Therefore, we incorporate a *copy* mechanism to handle such rare words problem for our comment generation tasks.

Specifically, we calculate $p_{cg} \in [0, 1]$, which determines whether to generate a word from the vocabulary or to copy the word directly from the input sequence, based on our previous attention distribution a_i^t:

$$p_{cg} = sigmoid(W_{eh}^T c_t^* + W_{sh}^T s_t + W_x x_t + b_{cg}) \tag{12}$$

Here W_{eh}, W_{sh}, W_x and b_{cg} are trainable model parameters. The final probability of decoding a word is specified by the mixture model:

$$P^*(Cword) = p_{cg} \cdot \sum a_i^t + (1 - p_{cg}) \cdot p(Cword). \tag{13}$$

Where $P^*(Cword)$ is the final distribution over the union of the vocabulary and the input sequence. For a word not in our output vocabulary, the probability will be $P^*(Cword) = 0$, and in such cases, our model will replace the **<unk>** token for out-of-vocabulary words with a word in the input sequence having the highest attention obtained using attention distribution a_i^t. The *copy* mechanism allows the model to locate a certain segment of the input sequence and puts that segment into the output sequence. p_{cg} is a soft switch to choose between generating a word from vocabulary or copying a word from the input sequence.

4.7 Incorporating a Coverage Mechanism

The *coverage* mechanism [56] is an improved method for the *attention* mechanism. In the standard attention mechanism, the decoder dynamically assigns attention weights to different parts of the input sequence at each decoding step. However, as the decoding progresses, the *attention* mechanism tends to focus on the same regions repeatedly, neglecting other parts of the input sequence. This can result in redundant or incomplete information being generated during the decoding process. To address this repetition problem, we incorporated the *coverage* attention mechanism into our model. Particularly, we maintain a word coverage vector *cov* which is the sum of

Table 1. The Statistics of Our Collected Smart Contracts

Contract	Function	Comment	Bytecode Length Avg	Coment Length Avg
54,739	1,323,554	565,403	6920.21	142.00

attention distributions over all previous decoder timesteps:

$$cov^t = \sum_{t'=0}^{t-1} a^{t'}. \tag{14}$$

Here, cov^t is a distribution over input tokens that represents the degree of coverage that those tokens have received from the *attention* mechanism so far. Since no word is generated before timestamp 0, $cov^0 0$ will be a zero vector. The update equation 7 is now modified to be:

$$e_i^t = v^t \tanh\left(W_{cv} cov_i^t + W_{eh} h_i + W_{sh} s_t + b_{att}\right) \tag{15}$$

Here, W_{cv} are trainable parameters that ensure the *attention* mechanism's current decision is informed by a reminder of its previous decisions. The *coverage* mechanism allows our model to solve the word repetition problem in the output sequence. The *coverage* mechanism ensures that the *attention* mechanism's current decision is informed by a reminder of its previous decisions (summarized in cov^t). This should make it easier for the *attention* mechanism to avoid repeatedly attending to the same locations, and thus avoid generating repetitive text. Following the incorporation of the *copy* and *coverage* mechanism in our sequence-to-sequence architecture, the final loss function will be:

$$Loss = \frac{1}{T} \sum_{t=0}^{T} logP^*(Cword_t) + \lambda L_{cov} \tag{16}$$

where λ is a reweighted hyperparameter and the coverage loss L_{cov} is defined as:

$$L_{cov} = \sum_i min(a_i^t, cov_i^t) \tag{17}$$

Once the model is trained, we do inference using a beam search. The beam search is parametrized by the possible paths number k. The inference process stops when the model generates the `END` token which stands for the end of the sentence.

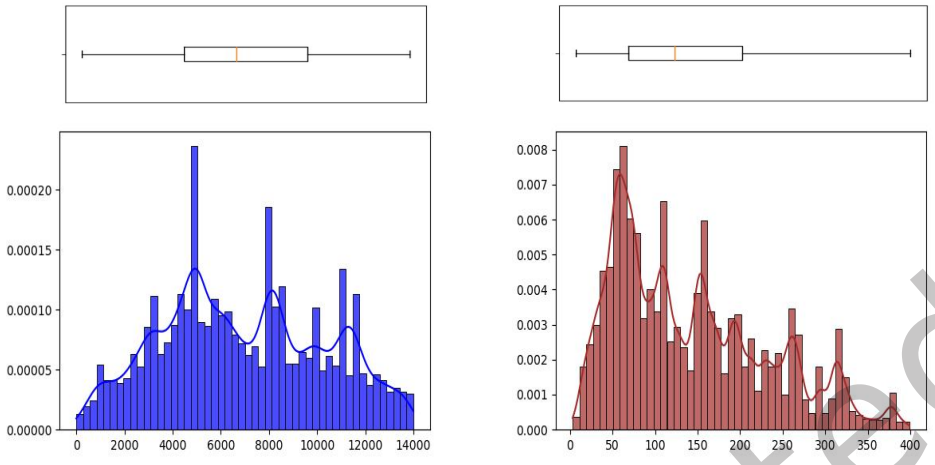
5 EVALUATION

5.1 DataSet Preparation

In this research, we reuse the raw smart contract dataset provided by Chen et al. [12], which contains 54,739 verified smart contracts. We describe how we prepared the dataset for our bytecode comment generation as follows.

5.1.1 ***Data Collection.*** For each verified smart contract, we crawled its source code and EVM bytecode from Etherscan [3]. As a result, we obtained 54,739 ⟨srccode, bytecode⟩ pairs of verified smart contracts.

5.1.2 ***Data Preprocessing.*** We exploit the Solidity-parser to parse smart contract source code and extract functions within each collected smart contract. We define several regular expressions to extract smart contract comments for functions. Following that, we utilize the `evm_cfg_builder` tool [2] tool to generate CFG for each function from EVM bytecode. Our dataset now is constructed as ⟨fucntion srccode, function bytecode, function comment⟩ triplets. Considering that duplicated data samples between the training set and testing set can mislead



(a) Bytecode length distribution (b) Comment length distribution

Fig. 5. Length Distribution of The Training Data

Table 2. Data Splitting Statistics

Total	Tain	Validate	Test
30,742	24,594	3074	3074

the evaluation results. We deduplicate our dataset according to the unique hash value generated by CFG and comment. Finally, we obtain 30,742 ⟨function srccode, function bytecode, function comment⟩ data samples. We list the statistics in Table 1 and the length distribution of the bytecode and comment in Figure 5.

5.1.3 ***Data Splitting.*** We split the constructed data samples into three chunks: 80 percent of the triple samples are used for training, 10 percent are used for validation and the rest are held out for testing. The training set is used to adjust the parameters, while the validation set is used to minimize overfitting, and the testing set is used only for testing the final solution to confirm the actual predictive power of our model with optimal parameters. The number of training, validation, and test sets of data samples are shown in Table 2.

5.2 Evaluation Metrics

To demonstrate the effectiveness of SmartBT, we use two widely used metrics in comment and code generation tasks [33]:

5.2.1 ***BLEU.*** BLEU [43] is a precision-oriented measure, which measures the average n-gram precision on a set of reference sentences, with a penalty for overly short sentences. It is widely used in various tasks of automatic software engineering, such as API sequence generation [25], comment generation [30, 31, 62], and commit message generation [34]. BLEU calculates the similarity between the generated notice and references. The similarity is computed as the geometric mean of n-gram matching precision scores multiplied by a brevity penalty to prevent very short generated sentences. In addition, we introduce the smoothing function Smooth2 in

BLEU evaluation, which can improve the stability of BLEU scores and reflect translation quality more accurately. In this paper, we adopt BLEU-1, BLEU-2, BLEU-3, and BLEU-4 scores.

5.2.2 ***ROUGE.*** ROUGE [39] is a widely used recall-oriented measure in summarization tasks. It evaluates the overlap of *n*-grams between system-generated summaries and reference sentences. ROUGE-1 and ROUGE-2 measure unigram and bigram overlaps, while ROUGE-L captures in-sequence matches reflecting sentence-level word order. To evaluate different models, we consider ROUGE-1, ROUGE-2, and ROUGE-L scores.

5.3 Training Details

We implement SmartBT on top of Pytorch. Both token embeddings and hidden size are set to 256 dimensions. All parameters are optimized using Adam [35] with the initial learning rate of 0.0005. Following Vaswani et al. [57], we increase the learning rate linearly for the first 4000 steps (i.e., warmup steps) and decrease it thereafter proportionally to the inverse square root of the step number. During the training, the batch size is set to 32. To mitigate overfitting, we exploit the dropout mechanism and set the dropout rate as 0.1. We set the maximum length of the encoder to 200 and the maximum length of the decoder to 50. Training runs for 50 epochs. We conduct our experiments on a Linux server with an NVIDIA GeForce RTX 2080Ti GPU having 10 GB memory.

6 EXPERIMENT RESULTS

In this study, we aim to answer the following research questions:

- *RQ1:* How effective is our SmartBT for generating smart contract comments from bytecode?
- *RQ2:* How effective is the IR component with different retrieval methods?
- *RQ3:* How effective is our use of *attention* mechanism, *copy* mechanism and *coverage* mechanism under automatic evaluation?
- *RQ4:* How effective is our SmartBT under different IR settings?
- *RQ5:* How effective are baseline models augmented with IR component?
- *RQ6:* How effective are LLMs for generating smart contract comments from bytecode?
- *RQ7:* How effective is our SmartBT under human evaluation?

6.1 RQ1. **SmartBT** Overall Effectiveness

6.1.1 Experimental Setup. In this RQ, we want to investigate how effective our approach and baseline methods are for generating smart contract comments from bytecode. In particular, all models are trained (i.e., SmartBT) and fine-tuned (i.e., CodeT5 and PLBART) with our training set. Then the models (including SmartBT and baselines) with their best performance on the validation set are used to report final comparison results. For each smart contract function in our test set, we use the intermediate representation of bytecode, i.e., CFG sequence, as model inputs (including SmartBT and baseline methods). Then SmartBT and baseline methods generate corresponding function comments as outputs based on the inputs. We then calculate the BLEU and ROUGE scores between the generated comments and ground truth comments for comparison purposes.

6.1.2 Baselines. Since no prior work focused on contract comment generation based on bytecode, we chose several commonly used baselines for this task. The following baselines are adopted in this study:

- **IR**: The IR stands for information retrieval baseline. For a given bytecode, it retrieves a function comment that is closest to the input bytecode from the training set. Since directly comparing bytecode is difficult, we calculate the similarity between the CFG sequence of two different bytecodes, and the most similar comment is fetched as the IR method result. In this study, we employ widely used BM25 [46] as our information retrieval algorithm to perform the information retrieval task.

- **CodeT5**: CodeT5 [61] is a unified pre-trained encoder-decoder Transformer model. CodeT5 better leverages the code semantics conveyed from the developer-assigned identifiers. CodeT5 builds on the T5 architecture [45] that supports both code understanding and generation and allows for multi-task learning. CodeT5 outperforms prior methods in code generation tasks including PL-NL, NL-PL and PL-PL. In this work, we use CodeT5 (CodeT5-small and CodeT5-base) as a baseline to perform our comment generation tasks. Considering it is difficult for CodeT5 to directly learn from bytecode, we use CFG sequences as inputs for CodeT5. We then fine-tuned the pre-trained CodeT5 model with our training set on the code summarization task, using CFG sequences as inputs and target corresponding comments as outputs. We fine-tuned CodeT5 for 50 epochs and the model with the highest BLEU-4 score was chosen for evaluation.
- **PLBART**: PLBART [5] is a bidirectional and autoregressive transformer pre-trained on unlabeled data across PL (Programming Language) and NL (Natural Language) to learn multilingual representations. PLBART is pre-trained on an extensive collection of Java and Python functions and NL text via denoising autoencoding. PLBART outperforms or rivals state-of-the-art models on code summarization, code generation, and code translation tasks. Similarly, we use CFG sequences as PLBART inputs. We fine-tuned PLBART the same way of fine-tuning CodeT5, the best model on the validation set is used for final evaluation.

6.1.3 Evaluation Results. The evaluation results of SmartBT and aforementioned baselines are summarized in Table 3. From the table, we can observe the following points:

(1) **In general, the encoder-decoder architecture baselines (i.e., CodeT5 and PLBART) outperform IR based approach (i.e. BM25)**. For IR based approach, it retrieves the comments from the existing database according to similarity scores, which heavily relies on the presence of similar Control Flow Graphs (CFGs) and the degree of similarity between them, indicating that merely memorizing the training set is not enough for this task. In contrast to the IR-based approach, the encoder-decoder model uses the vector representation for tokens and internal states, semantic and structural information can be learned from these vectors by taking global context into consideration.
(2) **Notably, our IR-augmented model (i.e.,** SmartBT**) even achieves a better performance than the pre-trained transformer-based models (e.g., CodeT5 and PLBART)**. This is because these pre-trained language models are mostly pre-trained on the extensive collection of code corpus (e.g., source code and NL text), however, these pre-trained models were not designed to handle opcode (e.g., CFG sequence), which results in their suboptimal performance for our newly proposed task. Moreover, compared with source code, the gap between the bytecode/opcode and comments is even larger, posing a significant challenge for pre-trained models to capture their semantic relationships effectively. This is the reason why we introduce IR-augmented techniques for bridging the semantic gap between low-level bytecode and natural language comments.
(3) **Regarding BLEU score, our approach is significantly better than other baselines and achieves understandable results.** For example, it improves over PLBART on BLEU-4 by 22%. We attribute this to the following reasons: besides solely depending on structural inputs such as CFGs, our model also incorporates an IR-augmented module, the IR augmented module fetches comments from similar smart contracts. In other words, SmartBT combines the structural input (i.e., CFGs) and semantic input (i.e., similar comments) for constructing the contextual vectors, which signals that similar comments convey much valuable information when generating target comments.
(4) **Regarding ROUGE score, the advantage of our approach is also clear.** This is because our model is enhanced by a *copy* mechanism to handle rare-word problems as well as a *coverage* mechanism to eliminate meaningless repetitions. This further justifies the aforementioned mechanisms generally help when dealing with the comment generation task.

Table 3. Automatic Comment Generation Evaluation Using BLEU and ROUGE Scores (%)

Method	BLEU-1	BLEU-2	BLEU-3	BLEU-4	ROUGE-1	ROUGE-2	ROUGE-L
BM25	24.08	23.13	22.57	22.24	26.35	20.32	25.78
CodeT5-small	25.49	24.69	24.11	23.55	29.49	21.80	29.02
CodeT5-base	28.18	27.44	26.83	26.21	32.10	24.76	31.67
PLBART-base	30.03	29.01	27.93	27.24	36.01	27.80	35.46
SmartBT	**37.18**	**35.65**	**34.15**	**33.24**	**41.14**	**32.44**	**40.53**

Answer to RQ-1: How effective is our SmartBT for generating smart contract comments from bytecode? We conclude that our approach is effective for generating smart contract comments from the bytecode under automatic evaluation and surpasses baselines by a large margin.

6.2 RQ2. Effectiveness of Different IR Methods

6.2.1 Experimental Setup. In this RQ, we want to investigate how much performance improvement our approach can achieve by using the IR-augmented component. In particular, we selected several different methods as the information retrieval algorithm for IR-augmented component. We then used these different IR-augmented components to generate target comments and calculated the BLEU and ROUGE scores for comparison purposes.

6.2.2 Baselines. To fill the semantic gap between bytecode and comment, we introduce the IR (Information-retrieval) augmented module to fetch similar comments from contracts with similar code structures and logic. Intuitively, our IR augmented module can take any similarity matching methods. To verify the effectiveness of our using IR augmented component, we choose the following information retrieval methods for this research question.

- $\mathbf{IR_{None}}$: For this baseline, we remove the IR augmented component and keep the rest of SmartBT. In other words, we drop the semantic input (i.e., similar comment) of our model and only retain the structural input (i.e. CFGs). This baseline is denoted as $\mathbf{IR_{None}}$.
- **BOW:** Cosine similarity [47] is one of the most popular distance metrics used for comparing the similarity between two vectors. Regarding this baseline, we convert the CFG sequence into BOW (bag-of-words) vectors, then cosine similarity scores are computed between any two given CFGs, this baseline is denoted as **BOW**.
- **GraphCodeBERT:** Besides using the BOW to represent the CFG sequence, we also use GraphCodeBERT [26] to encode CFG into vector representations. In particular, for a given smart contract bytecode, its CFG sequence is fed into GraphCodeBERT to obtain the feature representation vector. Then the vector is used to fetch the most similar smart contract by calculating the cosine similarity scores between two embedding vectors.
- **SmartBT:** Our approach employs the widely used BM25 as our information retrieval algorithm. BM25 aims to provide accurate and relevant search results by scoring documents based on their term frequencies and document lengths. It follows the probabilistic retrieval framework, which assumes that relevant and non-relevant documents follow different statistical distributions.

6.2.3 Evaluation Results. The evaluation results of SmartBT and aforementioned baselines are summarized in Table 4. From the table, it can be seen:

Table 4. Effects of Different Information Retrieval Methods on Our Task: BLEU and ROUGE Scores (%)

Method	BLEU-1	BLEU-2	BLEU-3	BLEU-4	ROUGE-1	ROUGE-2	ROUGE-L
IR_{None}	31.18	30.19	29.36	28.90	34.61	27.46	34.03
Bag-Of-Words	33.22	32.51	31.11	30.00	37.21	30.47	36.89
GraphCodeBERT	33.66	32.96	31.65	30.66	39.18	32.80	38.85
SmartBT	**37.18**	**35.65**	**34.15**	**33.24**	**41.14**	**32.44**	**40.53**

(1) **The IR augmented module is effective in enhancing the effectiveness of our model and contributes to the overall performance.** Compared with IR_{None}, there is an improvement in terms of all evaluation metrics after adding IR augmented module, regardless of the information retrieval algorithm. For example, using **BOW** and **GraphCodeBERT** to fetch relevant comments as inputs, the BLEU-4 score has improved by 3.8% and 6% respectively. The experimental results indicate that the encoder-decoder neural network merely using structural inputs (i.e., CFGs) is unable to bridge the semantic gap between bytecode and comments, verifying the importance and necessity of using semantic inputs (i.e., similar comments) via using IR augmented component.

(2) **SmartBT, using the BM25 algorithm, achieved the best performance among other information retrieval algorithms.** We attribute this to the following advantages of the BM25 algorithm: (a) Term saturation: BM25 incorporates a term saturation function, this function mitigates the impact of excessively high term frequencies. (b) Dynamic ranking: BM25 adjusts its ranking based on the distribution of terms within the collection, making it more adaptable to different types of documents and queries. (c) Effective for long queries: BM25 is effective in handling long CFG sequences as it addresses the issue of term saturation and considers the overall CFG sequence length.

Answer to RQ-2: How effective is the IR component with different methods? We conclude that the inclusion of the IR augmented module significantly enhances our model's effectiveness and contributes to its overall performance. The BM25 performs best for fetching relevant comments from smart contracts and provides semantic inputs for **SmartBT**.

6.3 RQ3. Effectiveness of Using *Copy* and *Coverage* Mechanisms

6.3.1 Experimental Setup. In this RQ, we conduct an ablation experiment to verify the effectiveness of using the *copy* mechanism and *coverage* mechanism within our model. In particular, we removed the *copy* mechanism and *coverage* mechanism one by one, and then we calculated the BLEU and ROUGE scores between the generated comments and ground truth comments for comparison purposes.

6.3.2 Baselines. When constructing **SmartBT**, we added an *attention* mechanism, a *copy* mechanism, and a *coverage* mechanism to our IR augmented encoder-decoder architecture. The *attention* mechanism is often regarded as a basic mechanism incorporated within the sequence-to-sequence model. The *copy* mechanism is used to handle rare-word problems in our generation process and the *coverage* mechanism is used to eliminate meaningless repetitions while decoding. To verify the effectiveness of the above mechanisms, to be more specific, we compare our **SmartBT** with several of its incomplete variants:

- **Model**$_{atten}$: This baseline removes the *copy* and *coverage* mechanisms from our approach while keeping the basic *attention* mechanism as the basic model.

- **Model**$_{atten+copy}$: This baseline removes the *coverage* mechanism from our approach while keeping the *attention* mechanism and the *copy* mechanism.
- **Model**$_{atten+coverage}$: This baseline removes the *copy* mechanism from our approach while keep the *attention* mechanism and *coverage* mechanism.
- **SmartBT**: It is our current work which considers all the three mechanisms.

Table 5. Ablation Evaluation of Different Mechanisms Using BLEU and ROUGE Scores (%)

Model	BLEU-1	BLEU-2	BLEU-3	BLEU-4	ROUGE-1	ROUGE-2	ROUGE-L
Model$_{atten}$	28.37	27.23	26.32	25.83	32.01	24.29	31.39
Model$_{atten+copy}$	32.29	31.18	30.28	29.81	35.95	28.32	35.38
Model$_{atten+coverage}$	30.11	28.84	27.90	27.36	33.32	25.59	32.70
SmartBT	**37.18**	**35.65**	**34.15**	**33.24**	**41.14**	**32.44**	**40.53**

6.3.3 Evaluation Results. The evaluation results of the ablation study are summarized in Table 5. From the table, we can observe the following points:

(1) By comparing the results of **Model**$_{atten}$ with **Model**$_{atten+copy}$ and **Model**$_{atten+coverage}$, we can measure the performance improvements achieved due to the incorporation of *copy* mechanism and *coverage* mechanism respectively. **It is clear that better performance can be achieved by solely adding *copy* or *coverage* mechanism to the attention based model**. This signals that both *copy* and *coverage* mechanism do have contributions to the overall performance improvements.
(2) By comparing the results of our approach and each of the variant model, **we can see that no matter which type of mechanism we removed, there is a drop overall in every evaluation measure metric and does hurt the performance of our model.** Particularly, when comparing the our model **SmartBT** with **Model**$_{atten}$, it drops almost 10% of the overall automatic evaluation scores. This verifies the importance and effectiveness of these incorporated mechanisms.
(3) To gain insights into these mechanisms, we further illustrate an example in Figure 6 from the ablation analysis to show the effect of employing *copy* and *coverage* mechanism. From the figure, we can see that: (a) After introducing the *copy* mechanism, the model can **copy** relevant tokens from our IR augmented module to the generated comment. For example, the words `beneficiary` and `funders` (colored in red) are copied directly from the IR-augmented component to the target outputs. (b) Repetition is a common problem for attentional sequence-to-sequence models, meaningless repeated words are produced during the generation process (highlighted with green color). The *coverage* mechanism is effective for discouraging such repetitions by quantitatively emphasizing the coverage of sentence words while decoding. For example, the word "`function`" has been meaningless repeated twice, employing *coverage* mechanism can effectively eliminate such repetitions.

Answer to RQ-3: How effective is our use of *attention* mechanism, *copy* mechanism and *coverage* mechanism under automatic evaluation? We conclude that all the incorporated mechanisms do have contributions to the overall performance and are effective and helpful in enhancing the performance of **SmartBT**.

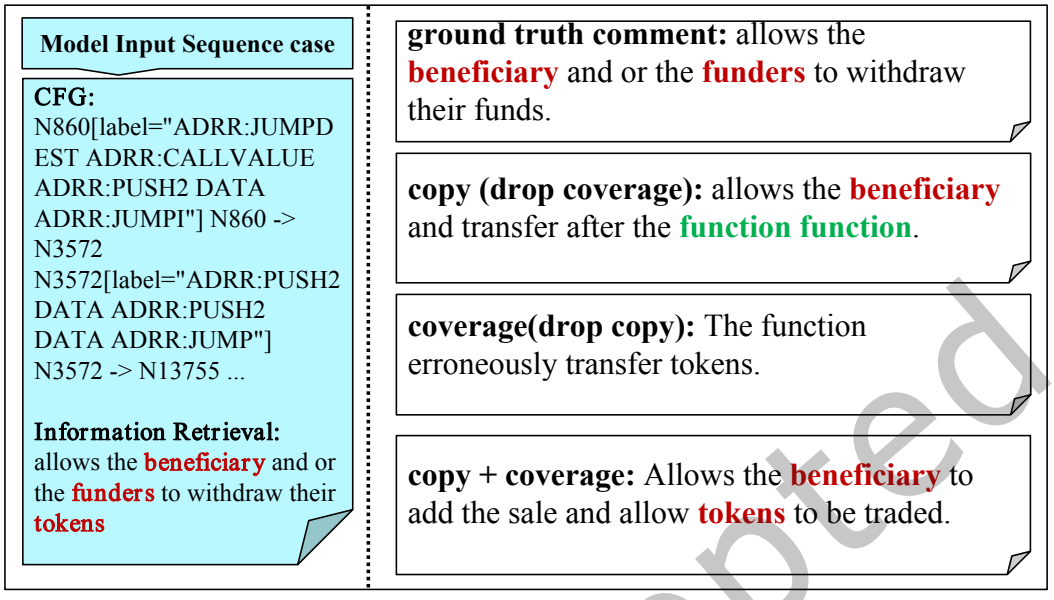


Fig. 6. Ablation Analysis Example

6.4 RQ4. Effectiveness of Different IR Settings

6.4.1 Experimental Setup. In this RQ, we want to investigate how **SmartBT** performs under different IR augmented module settings. In particular, we aim to explore the optimal IR-augmented module settings for our task. Based on the findings of RQ3, we vary the number of retrieved comments for our model inputs and compare generated results under different module settings.

6.4.2 Baselines. The key hyperparameter of our IR augmented module is the number of similar comments retrieved for **SmartBT** inputs. In this RQ, to investigate the optimal settings for the IR augmented module, we vary the number of retrieved comments to construct the baselines, denoted as $\mathbf{IR}_k$, where k represents the number of retrieved comments. In other words, $\mathbf{IR}_k$ retrieves k most relevant comments from our database and concatenates these comments as semantic inputs for our model. The k is varied in $[0, 1, 3, 5]$ for this study, where $k = 1$ represents our current model **SmartBT**.

Table 6. Effects of Different Settings of IR Component Using BLEU and ROUGE Scores (%)

Model	BLEU-1	BLEU-2	BLEU-3	BLEU-4	ROUGE-1	ROUGE-2	ROUGE-L
$\mathbf{IR}_{k=0}$	31.18	30.19	29.36	28.90	34.61	27.46	34.03
$\mathbf{IR}_{k=3}$	26.23	25.33	24.16	23.22	30.54	22.48	30.01
$\mathbf{IR}_{k=5}$	23.78	22.13	21.03	20.34	27.60	17.41	26.91
SmartBT	**37.18**	**35.65**	**34.15**	**33.24**	**41.14**	**32.44**	**40.53**

6.4.3 Evaluation Results. The evaluation results are listed in Table 6, it can be seen that:

(1) **Adding semantic input for our model greatly improves the overall performance of our model.** Comparing our approach to the case of $\mathbf{IR}_{k=0}$ (where no similar comments are retrieved for semantic inputs), our model shows an improvement of nearly 19% in overall evaluation metrics. This demonstrates the significant impact of our IR-augmented module and the appropriate value of k.
(2) **Regarding the number of retrieved comments, not the more the better.** When adding more retrieved comments to our semantic inputs (e.g., top3 and top5 similar comments), we can find that the BLEU and ROUGE scores significantly drop. This is because a larger k may introduce more noise into our semantic inputs with more irrelevant references, which can increase the difficulty of generating correct comments.
(3) **By analyzing the performance of our approach with respect to different k, we notice that SmartBT achieves its optimal performance when $k = 1$.** The overall performance trend of our model decreases as k increases, which supports our concern that larger k settings introduce more noise and bring bigger challenges for our task. We thus set $k = 1$ for constructing our IR augmented module.

Answer to RQ-4: How effective is our SmartBT under different settings of IR component? We conclude that our **SmartBT** performs best when the top 1 relevant comment is retrieved for our semantic input, further increasing the k can introduce more noise and bring bigger challenges for our task.

6.5 RQ5. Effectiveness of Adding IR Component on Baselines

6.5.1 Experimental Setup. In this RQ, we aim to explore the impact of IR component on enhancing the performance of baseline models. Specifically, we add the same IR component incorporated within our model to baselines. For a fair comparison, the information retrieval methods (i.e., BM25) and its settings (i.e., number of retrieving comments) are exactly the same with SmartBT. Following that, we augmented baselines with IR component to generate target comments and calculate the BLEU and ROUGLE scores for evaluation.

6.5.2 Baselines. In RQ1, we adopted the Pre-trained Language Models, i.e., CodeT5 (small and base models) and PLBART, as baselines for comparison. We thus augment the above pre-trained models with the IR component, denoted as *CodeT5-small+IR*, *CodeT5-base+IR* and *PLBART-base+IR* respectively. We provide the performance of each model without an IR component for easy reference. In this research question, we also provide the parameter size as an indicator for reference.

6.5.3 Evaluation Results. The experimental results are shown in Table 7, from the table we can observe the following points:

(1) By comparing the models augmented with IR components and without IR components, **it is clear that IR augmented models exhibit significant improvements over their respective baseline models across all evaluation metrics, suggesting considerable advancements for introducing the IR component**. For example, after adding the IR component to *CodeT5-small* and *CodeT5-base* models, their BLEU scores have been improved by 43% and 48% respectively. The ROUGE scores also demonstrate substantial improvements, *CodeT5-small+IR* and *CodeT5-base+IR* models show a notable improvement of 38% and 43% respectively, while *PLBART-base+IR* model also achieves an improvement of 7%. Overall, the introduction of the IR component can greatly enhance the performance of baseline models, further verifying the effectiveness of the IR component in filling the semantic gap and improving the quality of generated comments.
(2) After adding the IR component, the *CodeT5-base+IR* model even outperforms our SmartBT in terms of BLEU and ROUGE scores, while **our model has its advantages regarding parameter sizes, which is much more lightweight and flexible**. Compared with our SmartBT, our model only costs 26M, while *CodeT5-base+IR* and *PLBART-base+IR* cost 220M and 1,100M respectively, which are 8 and 40 times larger

Table 7. Effects of IR Augmented Baselines Using BLEU and ROUGE Scores (%)

Method	BLEU-1	BLEU-2	BLEU-3	BLEU-4	ROUGE-1	ROUGE-2	ROUGE-L	# Parameters
CodeT5-small	25.49	24.69	24.11	23.55	29.49	21.80	29.02	60M
CodeT5-small+IR	36.53	34.29	33.01	32.22	40.71	32.90	40.17	60M
CodeT5-base	28.18	27.44	26.83	26.21	32.10	24.76	31.67	220M
CodeT5-base+IR	41.93	39.96	38.54	37.66	45.82	38.88	45.40	220M
PLBART-base	30.03	29.01	27.93	27.24	36.01	27.80	35.46	1100M
PLBART-base+IR	35.39	29.30	26.80	25.80	38.14	29.03	36.70	1100M
SmartBT	**37.18**	**35.65**	**34.15**	**33.24**	**41.14**	**32.44**	**40.53**	**26M**

than our model. Our SmartBT can be easily set up and deployed on developers' personal computers, it can perform inferencing with CPUs, without requiring GPUs. As a result, our proposed model is more lightweight and flexible according to different developers' computing resources.

Answer to RQ-5: How effective are baseline models augmented with IR component? The IR-augmented models show significant improvements over baseline models across all evaluation metrics, further verifying the effectiveness and importance of introducing IR component.

6.6 RQ6. The Effectiveness of LLMs on Our Tasks

6.6.1 Experimental Setup. The Large Language Models (LLMs) (e.g., ChatGPT) are widely used by developers nowadays and have demonstrated promising performance for code-related tasks, such as code generation, test generation, and code summarization [14, 40, 59, 66, 67]. In this RQ, we want to investigate whether the LLMs can be adapted to our task successfully. In particular, we aim to explore the ability of LLMs on our tasks in terms of two aspects: generating comments directly from smart contract bytecode and generating comments from CFG sequences. Since it is too expensive to conduct the experiments on our full test set (i.e., 3,074 cases), we randomly selected a statistically representative sample of 342 cases from our test set (with a 95% confidence level and 5% margin of error). Following that, we calculated the BLEU and ROUGE scores of LLMs and SmartBT and these 342 cases for comparison purposes.

6.6.2 Baselines. The great success of ChatGPT demonstrates the remarkable ability of large language models (LLMs) to comprehend human questions and assist in code-relevant tasks. In this RQ, we adopted the GPT-4 as the baseline to perform our task. GPT-4 is the newest LLM created by OpenAI, it is a large multimodal model (accepting image and text inputs and emitting text outputs), that exhibits human-level performance on various professional and academic benchmarks. Currently, in-context learning (i.e., ICL) has been widely used by researchers to elicit human knowledge and logical reasoning from LLMs to accomplish complicated tasks. We explore the performance of two variants of in-context learning (i.e., zero-shot learning and few-shot learning) on two scenarios (i.e., input Bytecode and CFG sequence) respectively. We set the temperature request parameter to 0.7, which is a commonly used empirical standard in many papers [27]. This setting is known to generate results that are relatively natural and exhibit a certain level of creativity.

- **Zero-shot Learning:** With the increasing ability of LLMs, in-context learning has shifted to a new paradigm, known as zero-shot learning, where LLMs make predictions by directly describing the desired output. Zero-shot prompting involves posing a question or task to the model without providing any specific context or examples. We apply zero-shot learning to bytecode and CFGs respectively, denoted as GPT-4-Zero-RBC and GPT-4-Zero-CFG.

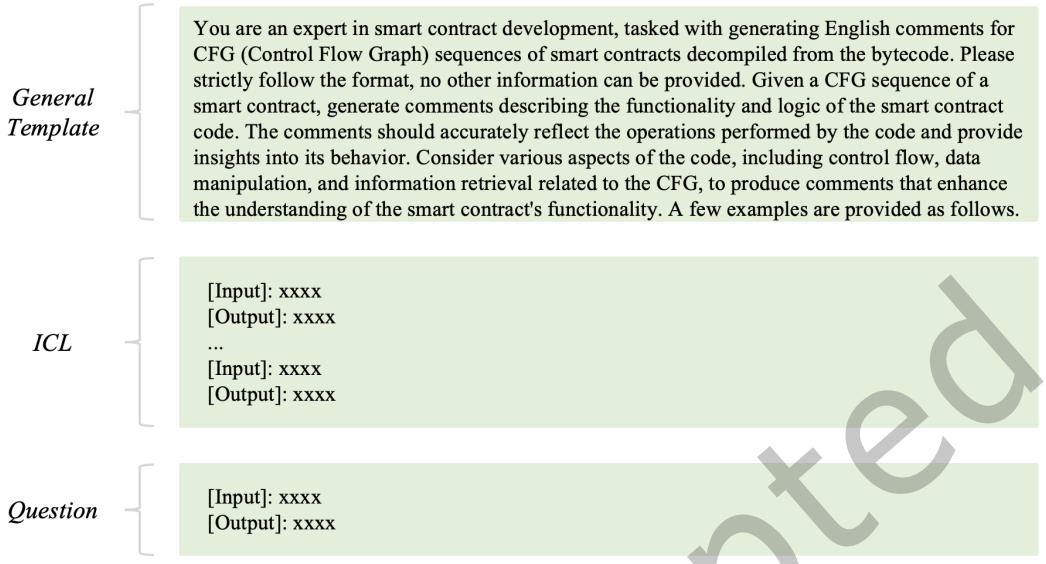


Fig. 7. An ICL Prompt Template Example

- **Few-shot Learning:** While zero-shot learning shows promising performance in various tasks by leveraging prior knowledge from training sources, it remains challenging to apply to unseen tasks. To overcome this challenge, few-shot learning is utilized to augment the context with a few examples of desired inputs and outputs (as shown in Fig. 7). Few-shot learning enables LLMs to recognize the input prompt syntax and patterns of output. We apply few-shot learning to bytecode and CFG input respectively, denoted as GPT-4-Few-RBC and GPT-4-Few-CFG.

Table 8. Effects of GPT-4 on Our Task Using BLEU and ROUGE Scores (%)

Method	BLEU-1	BLEU-2	BLEU-3	BLEU-4	ROUGE-1	ROUGE-2	ROUGE-L	# Parameters
GPT-4-Zero-RBC	10.56	8.00	7.00	6.41	14.24	4.33	13.63	1760B
GPT-4-Few-RBC	13.64	11.02	9.84	9.04	18.42	8.05	17.48	1760B
GPT-4-Zero-CFG	14.75	12.14	10.57	9.56	21.73	9.71	20.29	1760B
GPT-4-Few-CFG	17.01	14.19	12.70	11.74	24.18	11.32	22.74	1760B
SmartBT	**37.18**	**35.65**	**34.15**	**33.24**	**41.14**	**32.44**	**40.53**	**26M**

6.6.3 Evaluation Results. The experimental results of GPT-4 and SmartBT on the 342 test cases are summarized in Table 8, from the table, several points stand out:

(1) **It is difficult for LLM to generate comments from bytecode directly.** Compared with generating comments from CFG sequences, generating comments directly from bytecode inputs achieves the worst performance in terms of all evaluation metrics. Our experimental results reveal that LLMs can hardly understand the semantic meaning embedded within the bytecode, this is reasonable because even advanced

LLMs such as GPT-4, have not been pre-trained with bytecode data, thereby hindering them from generating target comments accurately and effectively.

(2) By comparing GPT-4's performance with zero-shot learning and few-shot learning, it is clear that **few-shot learning has its advantage over zero-shot learning.** For example, with few-shot learning, GPT-4 improves the BLEU-4 score over zero-shot learning by 41% and 22% regarding bytecode and CFG sequence respectively. This is because zero-shot learning relies solely on the model's preexisting knowledge to generate responses. When GPT-4 doesn't have specific knowledge triggered by the prompt, it may provide generic or unrelated responses. Few-shot learning prompts the GPT-4 with concrete examples, enhancing the model's understanding of the given task.

(3) SmartBT **outperforms GPT-4 by a large margin in terms of all evaluation metrics**. The suboptimal performance of GPT-4 on our tasks indicates that LLMs are not suitable for this task because they are not pre-trained with the bytecode datasets and/or designed for handling bytecode-relevant tasks. Compared with GPT-4, we introduce CFGs to capture the structural information of bytecode and IR-augmented components to capture the semantic information of bytecode. It would be interesting to explore the effectiveness of pre-training and fine-tuning LLMs with bytecode and/or developing LLMs particularly tailored to handle bytecode-related tasks, but this is beyond the scope of our current research and we plan to leave it for future work.

Answer to RQ-6: How effective are LLMs for generating smart contract comments from bytecode? We conclude that it is difficult for LLMs such as GPT-4 to generate comments from smart contract bytecode because LLMs are seldom pre-trained with bytecode data and can hardly generalize to this unseen task.

6.7 RQ7. Human Evaluation

6.7.1 Experimental Setup. Since automated evaluation indicators such as BLEU and ROUGE cannot really reflect the effects of generating comments, we conducted a user study to make more real evaluations. In this RQ, we investigate the effectiveness of our approach through human evaluation [18]. To be more specific, we invited participants to manually assess the quality of our generated comments in terms of the following two aspects:

- **Naturalness:** Naturalness evaluates the grammatical correctness and fluency of the generated comments. It assesses how well the comments read and flow in natural language, ensuring they are easily comprehensible to human users.
- **Relevance:** Relevance assesses the comments' alignment with the ground truth comment. It measures the degree of relevance and coherence between the generated and reference comments, indicating how well the generated comments capture the intended meaning and information present in the original code.

6.7.2 Baselines. The BM25, CodeT5 and PLBART are selected as baselines for human evaluation. Particularly, we invited 20 with 1-3 years of blockchain or smart contract experience and good English proficiency to evaluate the generated comments in the form of a questionnaire. An example of the questionnaire is demonstrated in Figure 8. We randomly sampled 25 ⟨generated comment, reference comment⟩ pairs from our evaluation dataset. To ensure the quality of the manual analysis, we divided the 20 participants into five groups, each group of participants will only rate 5 data samples independently. In other words, each data sample was rated by 4 independent evaluators. Each participant is required to compare the candidate comments with the ground truth comment and estimate their **Naturalness** and **Relevance** on a scale between 1 and 5 (5 is the best). During the annotation, participants are allowed to search the Internet for related information and unfamiliar concepts. Participants do not know which comment was generated by which approach.

Please rate each Candidate for Naturalness and Relevance from 1-5 (5 is the best) compared with the Ground Truth comment		Naturalness	Relevance
Ground Truth:	*Get the amount of allowed tokens to spend*		
Candidate 1:	*Approves and then calls the receiving contract*	3	1
Candidate 2:	*Returns the amount of tokens approved by the owner that can be transferred*	5	3
Candidate 3:	*ERC20 return allowance for given owner spender pair*	3	1
Candidate 4:	*Returns the amount of tokens approved by the owner that can be transferred to the spender's account*	5	5

Fig. 8. A User Study Case

Table 9. Human Evaluation

Method	N	Low_N	$Medium_N$	$High_N$	R	Low_R	$Medium_R$	$High_R$
BM-25	3.57	15%	31%	54%	1.74	85%	6%	9%
CodeT5	4.15	25%	22%	53%	3.54	18%	22%	60%
PLBART	3.66	18%	22%	60%	2.69	52%	19%	29%
SmartBT	**4.34**	**0%**	**12%**	**88%**	**3.72**	**18%**	**18%**	**64%**

6.7.3 Evaluation Results. After collecting responses from all evaluators, we regard a score of 1 and 2 as *low quality*, a score of 3 as *medium quality*, and a score of 4 and 5 as *high quality*, respectively. We then calculated the average scores and the proportion of each quality type (e.g., *low quality*, *medium quality* and *high quality*) respectively. The quality distribution and average score of naturalness and relevance across each method are presented in Table 9. From the table we can draw the following conclusions:

(1) **Regarding the naturalness score, SmartBT outperforms other methods by a large margin**. Notably, the proportion of low-quality comments is 0%, while the proportion of high-quality comments is 88%. This indicates that our approach excels in generating fluent and grammatically correct comments.
(2) **Regarding the relevance score, SmartBT also surpasses the other methods, indicating that the generated comments exhibit a higher degree of relevance to the comments written by developers.** For instance, the average relevance score of SmartBT is 3.72, which is the highest among all other methods. Furthermore, 64% of the generated comments are rated of high quality by participants. An example of this is demonstrated in Figure 8, candidate 4 ("*returns the number of tokens approved by the owner that can be transferred to the spender's account*") is the comment generated by our approach. Even the generated comment and human written comment do not share many common words, but they are semantically equivalent and rated as high relevance scores by evaluators.
(3) **In general, our model performs well across both dimensions.** The results of human evaluation are consistent with automatic evaluation results. The considerable proportion of high-quality comments generated by our approach with respect to the **Naturalness** and **Relevance** also reconfirms the effectiveness of our system.

Answer to RQ-7: How effective is our SmartBT under human evaluation? We conclude that under human evaluation, our approach outperforms other baselines in terms of both **Naturalness** and **Relevance** aspects.

7 THREATS TO VALIDITY

In this section, we discuss potential threats to the validity of our study, which are concerns related to the accuracy and generalizability of our findings.

Internal Validity: Threats to interval validity primarily are concerned with potential errors that may have occurred in our code implementation and research settings. To minimize such errors, we have conducted thorough inspections and fully tested our source code in both model design and automatic evaluation. The parameters of the baseline methods have been carefully tuned and their highest-performing configurations are reported for comparison. However, despite these efforts, there remains a possibility of unnoticed errors in our implementation. Considering such cases, we have published our source code and dataset to facilitate other researchers to replicate and extend our work.

External validity: Threats to external validity relate to the quality and generalizability of our dataset. Our dataset includes Solidity source code and their complied bytecode for smart contracts. There are other programming languages for smart contract development which are not considered in our study, we believe our results can generalize to other smart contract programming languages due to overall similarity in EVM bytecode representation. Future research will explore other smart contract languages' impact on code comment generation to enhance SmartBT's applicability. Considering smart contracts have a very high clone ratio on the Ethereum blockchain, we performed data deduplication in our data preprocessing. In particular, we deduplicated our dataset according to the unique hash value generated by a contract's CFG and its comment. In other words, we can make sure our training set doesn't overlap with our testing set. A more comprehensive evaluation setting is to split the data timewisely (e.g., using historical data for training and future data for testing), we will explore the timewise evaluation of our approach in our future work.

Construct validity: Threats to construct validity relate to issues that could affect the ability to draw correct conclusions about the relations between the treatment and outcome of an experiment. Evaluation metrics suitability is the primary issue. We used BLEU and ROUGE for automatic evaluation, which are widely used for evaluating grammatical correctness and relevance. However, semantic understanding might be limited due to diverse natural language expressions. Although we incorporate manual evaluation, it can be affected by evaluator attentiveness, language proficiency, and blockchain expertise. To address this, we select experienced participants proficient in English and blockchain and provide each participant with reasonable data samples and sufficient time for evaluation.

8 DISCUSSION

In this section, we discuss the adaptability of SmartBT to generate comments for other programming languages and the unique ability of SmartBT to generate comments for smart contracts.

8.1 Comment Generation from Bytecode

SmartBT is proposed for handling smart contracts where their source code is missing, under such situations, only bytecode is available on the Ethereum blockchain. Therefore, we design SmartBT to directly translate smart contract bytecode to natural language comments. **Notably, even though our approach is designed for smart contracts, it can be easily extended to other programming languages**. In general, SmartBT can be regarded as a general framework for generating comments from bytecode, which uses CFG to capture the structural information and the IR-augmented component to capture semantic information. It is an interesting research

direction to explore the effectiveness of SmartBT on other programming languages (e.g., Java), but it is beyond the scope of this research. Generating comments from bytecode not only benefits smart contract developers but also provides practical implications for other programming languages. For example, when developers face the task of decompilation (e.g., Android apps with only bytecode), the comments of the bytecode can provide a bird's eye view of the system's functionalities.

Recently, researchers have investigated of generating code comments from bytecode for popular programming languages, such as Java. For example, Huang et al. [32] first proposed a method named BCGen to generate comments for Java bytecode in 2022. Similarly, they converted the bytecode into CFGs and built the neural language model to learn from the CFGs and token sequences. Our SmartBT differs from the existing studies of generating code summaries from bytecode for general languages in terms of the following aspects: 1) Our research focuses on the EVM bytecode, this is because Solidity is specifically designed for writing smart contracts and is most widely used by smart contract developers. As far as we know, there is no ready-made dataset available for smart contracts and EVM bytecode, our work builds the first large dataset for this task. 2) Previous studies find that smart contracts have a relatively high clone ratio (e.g., 90%), much higher than the code clone ratio in traditional software projects [17]. Inspired by this finding, we introduced the IR augmented component to retrieve the similar function's comment to assist the target comment generation. The experimental results verify the effectiveness of using IR components to enhance the overall performance of our approach.

8.2 Comment Generation for Smart Contract

The comment generation task has been widely explored by software engineering researchers, however, there are only a few studies that investigated comment generation task for smart contracts [29, 51, 68]. Among them, two representative tools are closely related to our work, i.e., SmartDoc [29] and Stan [38]. Hu et al. [29] first introduce the task of generating descriptions for smart contracts from source code, they propose SmartDoc to generate user notices for smart contract functions. Our SmartBT shares the same architecture with SmartDoc by using the sequence-to-sequence learning of neural network models. However, different from SmartDoc which targets on smart contract source code, our approach first focuses on smart contract bytecode. However, only 13% of smart contract source code are available, which means SmartDoc can only be applied to a small number of smart contracts, while our SmartBT can support all smart contracts because their bytecode are all available on-chain. Moreover, compared with the source code, the gap between comments and natural language comments is even larger, therefore we introduced the IR augmented component to bridge this semantic gap.

Li et al. [38] proposed a model named Stan to generate descriptions for bytecode of smart contracts. Compared with Stan, SmartBT is more general. Stan describes every smart contract interface from four aspects (i.e., functionality description, usage description, behavior description, and payment description). SmartBT aims to generate comments for smart contract functions, since comments are natural language descriptions written by developers, the comments generated by our approach are more general to describe the smart contract. Moreover, Stan requires to analyze the extra metadata and adopts symbolic execution techniques to generate intermediate information, while SmartBT is an end-to-end model that only acquires the runtime bytecode of a smart contract as input and automatically outputs the natural language comment. Therefore SmartBT is more lightweight and flexible compared with Stan.

9 RELATED WORK

We employ the deep learning model to generate natural language comments for the bytecode of the smart contract, which are mainly related to the following three main aspects.

9.1 Code Comment Generation

Code comment generation is the most relevant task which aims to generate natural descriptions for code snippets. The Code comment generation task has been studied by a lot of software engineering researchers. Manually-crafted templates [41, 42, 52], IR techniques [28, 37, 63, 64], and neural network models [7, 30, 58, 62] are widely used in automatic comment generation. For example, Sridhara et al. [52] propose to construct Software Word Usage Model (SWUM) to select relevant keywords from source code and then leverage them to construct natural language descriptions from defined templates. Haiduc et al. [28] exploit two IR techniques, Vector Space Model (VSM) and LSI, to analyze methods and classes in Java projects and generate short descriptions for them. Iyer et al. [33] first propose to utilize the encoder-decoder framework to generate comments, in which the encoder is token embeddings of source code and the decoder is an LSTM. The experimental results on C# and SQL comment generation illustrate that neural networks perform better than traditional techniques.

Despite the availability of various automated comment generation studies and tools [6, 24, 69], there are few tools specifically designed for generating comments on smart contract code [29, 51, 68]. Hu et al. [29] first introduce the task of generating descriptions for smart contracts from source code. However, generating smart contract comments from EVM bytecode has never been investigated. To the best of our knowledge, our work is the first research to explore the possibility of generating smart contract comments from EVM bytecode, and our extensive evaluation shows the effectiveness of our tool for this newly proposed task.

9.2 Smart Contract Bytecode Analysis

Smart contract bytecode has been investigated in various tasks, including smart contract vulnerability detection [9, 13], smart contract classification [48, 50], and code similarity detection [71]. For example, Chen et al. [13] proposed DefectChecker, a symbolic execution-based approach and tool to detect eight contract defects that can cause unwanted behaviors of smart contracts on the Ethereum blockchain platform. DefectChecker can detect contract defects from smart contracts' bytecode. Ashizawa et al. [9] proposed Eth2Vec, a machine-learning-based static analysis tool for vulnerability detection in smart contracts. Eth2Vec automatically learns features of vulnerable EVM bytecodes with tacit knowledge through a neural network for natural language processing. Shi et al. [50] proposed a novel bytecode-based classification approach to effectively classify smart contracts of blockchain platforms. There are also techniques investigating how to decompile smart contract bytecode and generate Solidity source code. For example, Grech et al [21] presented Gigahorse, which is a reverse compiler that decompiles the smart contract EVM bytecode into high-level three-address code representation. Following that, they proposed Elipmoc [22] to further improve Gigahorse by integrating high-precision algorithms and design decisions that target a balance of precision and scalability. Suiche et al. [53] proposed Porosity, which is able to generate readable Solidity syntax contracts and enable static or dynamic analysis on these compiled contracts. However, even with these decompilers, it is still not easy for users to grasp the semantic information of the contract, not to mention the potential misleading due to decompilation errors.

Different from previous studies that focus on smart contract vulnerability detection from bytecode, our work first introduces the task of generating descriptions for smart contracts from EVM bytecode. Considering more than 90% of smart contract source codes are not available on the blockchain, while their EVM bytecode is always accessible, our tool complements the existing smart contract comment generation tools by using bytecode.

9.3 Software Engineering on Smart Contract

Software engineering researchers have investigated smart contracts for different software engineering tasks, such as smart contract clone detection [11, 15–17, 36], smart contract code searching [49], smart contract program repair [60, 70]. For example, Gao et al. [17] proposed a model, named SmartEmbed to detect code clones and clone-related bugs in smart contracts by using structural code embedding techniques. Shi et al. [49] proposed a

Multi-Modal Smart contract Code Search (MM-SCS) model for semantic code search with smart contracts. Their model can capture the data-flow and control-flow information from source code as well as semantic features. Yu et al. [70] proposed the first general-purpose automated smart contract repair approach that is also gas-aware. Their program repair model is search-based and searches among mutations of the buggy contract. Compared with the aforementioned studies, our research focuses on smart contract code comprehension and maintenance, our approach can greatly benefit smart contract developers and end users in understanding the functionality and logic of smart contracts from EVM bytecode instead of source code.

10 CONCLUSION

In this work, we first introduce the task of generating smart contract function descriptions from their EVM bytecode. We have proposed a novel model, named SmartBT, for translating bytecode to function comment automatically. SmartBT employs IR augmented module to fill the semantic gap between bytecode and natural language comments and adopts the encoder-decoder neural network to learn structural information from smart contract bytecode. We have conducted extensive experiments to verify the effectiveness of our approach, and SmartBT gets remarkable automatic evaluation scores and understandable human evaluation.

11 ACKNOWLEDGMENTS

This research/project is supported by the National Key Research and Development Program of China (No. 2021YFB2701102), the National Science Foundation of China (No.62372398, No. 62202341, No.72342025, and U20A20173), the Fundamental Research Funds for the Central Universities (No. 226-2022-00064). This research is partially supported by the Shanghai Sailing Program (23YF1446900). This research is partially supported by the Starry Night Science Fund of Zhejiang University Shanghai Institute for Advanced Study, Grant No. SN-ZJU-SIAS-001. This research is partially supported by the Ningbo Natural Science Foundation (No. 2023J292). This research is also supported by the advanced computing resources provided by the Supercomputing Center of Hangzhou City University. The authors would like to thank the reviewers for their insightful and constructive feedback.

REFERENCES

[1] 2018. *The Biggest ICO Scams of 2018.* Retrieved July 10, 2023 from https://bitcoinafrica.io/2018/09/27/biggest-ico-scams/
[2] 2019. *EVM CFG Builder GitHub Repository.* Retrieved July 10, 2023 from https://github.com/crytic/evm_cfg_builder
[3] 2023. *Etherscan Ethereum Blockchain Explorer.* Retrieved July 10, 2023 from https://etherscan.io/
[4] 2023. *Zellic.* Retrieved July 10, 2023 from https://www.zellic.io/
[5] Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Unified pre-training for program understanding and generation. *arXiv preprint arXiv:2103.06333* (2021).
[6] Miltiadis Allamanis, Hao Peng, and Charles Sutton. 2016. CodeNN: Neural Code Representation Learning for Code Summarization. In *Proceedings of the 40th International Conference on Software Engineering (ICSE).*
[7] Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. 2018. code2seq: Generating Sequences from Structured Representations of Code. In *International Conference on Learning Representations.*
[8] Anonymous. 2023. SmartBT Dataset. Retrieved July 28, 2023 from https://figshare.com/s/e5f3f9371ca67a63c122
[9] Nami Ashizawa, Naoto Yanai, Jason Paul Cruz, and Shingo Okamura. 2021. Eth2Vec: learning contract-wide code representations for vulnerability detection on ethereum smart contracts. In *Proceedings of the 3rd ACM international symposium on blockchain and secure critical infrastructure.* 47–59.
[10] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. 2014. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473* (2014).
[11] Jiachi Chen, Jiang Hu, Xin Xia, David Lo, John Grundy, Zhipeng Gao, and Ting Chen. 2024. Angels or demons: investigating and detecting decentralized financial traps on ethereum smart contracts. *Automated Software Engineering* 31, 2 (2024), 63.
[12] Jiachi Chen, Xin Xia, David Lo, and John Grundy. 2021. Why do smart contracts self-destruct? investigating the selfdestruct function on ethereum. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 31, 2 (2021), 1–37.

[13] Jiachi Chen, Xin Xia, David Lo, John Grundy, Xiapu Luo, and Ting Chen. 2021. Defectchecker: Automated smart contract defect detection by analyzing evm bytecode. *IEEE Transactions on Software Engineering* 48, 7 (2021), 2189–2207.
[14] Zhenlong Dai, Chang Yao, WenKang Han, Yuanying Yuanying, Zhipeng Gao, and Jingyuan Chen. 2024. MPCoder: Multi-user Personalized Code Generator with Explicit and Implicit Style Representation Learning. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 3765–3780.
[15] Zhipeng Gao. 2020. When deep learning meets smart contracts. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. 1400–1402.
[16] Zhipeng Gao, Vinoj Jayasundara, Lingxiao Jiang, Xin Xia, David Lo, and John Grundy. 2019. Smartembed: A tool for clone and bug detection in smart contracts through structural code embedding. In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 394–397.
[17] Zhipeng Gao, Lingxiao Jiang, Xin Xia, David Lo, and John Grundy. 2020. Checking smart contracts with structural code embedding. *IEEE Transactions on Software Engineering* 47, 12 (2020), 2874–2891.
[18] Zhipeng Gao, Xin Xia, John Grundy, David Lo, and Yuan-Fang Li. 2020. Generating question titles for stack overflow from mined code snippets. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 29, 4 (2020), 1–37.
[19] Zhipeng Gao, Xin Xia, David Lo, and John Grundy. 2020. Technical Q8A site answer recommendation via question boosting. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 30, 1 (2020), 1–34.
[20] Zhipeng Gao, Xin Xia, David Lo, John Grundy, Xindong Zhang, and Zhenchang Xing. 2023. I know what you are searching for: Code snippet recommendation from stack overflow posts. *ACM Transactions on Software Engineering and Methodology* 32, 3 (2023), 1–42.
[21] Neville Grech, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis. 2019. Gigahorse: thorough, declarative decompilation of smart contracts. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 1176–1186.
[22] Neville Grech, Sifis Lagouvardos, Ilias Tsatiris, and Yannis Smaragdakis. 2022. Elipmoc: advanced decompilation of ethereum smart contracts. *Proceedings of the ACM on Programming Languages* 6, OOPSLA1 (2022), 1–27.
[23] Jiatao Gu, Zhengdong Lu, Hang Li, and Victor OK Li. 2016. Incorporating copying mechanism in sequence-to-sequence learning. *arXiv preprint arXiv:1603.06393* (2016).
[24] Xiaodong Gu, Kelvin Guu, and Geoffrey Hinton. 2020. CodeBERT: A Pre-trained Model for Programming and Natural Languages. In *Conference on Empirical Methods in Natural Language Processing (EMNLP)*.
[25] Xiaodong Gu, Hongyu Zhang, Dongmei Zhang, and Sunghun Kim. 2016. Deep API learning. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 631–642.
[26] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, et al. 2020. Graphcodebert: Pre-training code representations with data flow. *arXiv preprint arXiv:2009.08366* (2020).
[27] Taicheng Guo, Kehan Guo, Bozhao Nan, Zhenwen Liang, Zhichun Guo, Nitesh V. Chawla, Olaf Wiest, and Xiangliang Zhang. 2023. What can Large Language Models do in chemistry? A comprehensive benchmark on eight tasks. arXiv:2305.18365 [cs.CL]
[28] Sonia Haiduc, Jairo Aponte, Laura Moreno, and Andrian Marcus. 2010. On the use of automated text summarization techniques for summarizing source code. In *2010 17th Working Conference on Reverse Engineering*. IEEE, 35–44.
[29] Xing Hu, Zhipeng Gao, Xin Xia, David Lo, and Xiaohu Yang. 2021. Automating user notice generation for smart contract functions. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 5–17.
[30] Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. 2018. Deep code comment generation. In *2018 IEEE/ACM 26th International Conference on Program Comprehension (ICPC)*. IEEE, 200–20010.
[31] Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. 2020. Deep code comment generation with hybrid lexical and syntactical information. *Empirical Software Engineering* 25, 3 (2020), 2179–2217.
[32] Yuan Huang, Jinbo Huang, Xiangping Chen, Kunning He, and Xiaocong Zhou. 2023. BCGen: a comment generation method for bytecode. *Automated Software Engineering* 30, 1 (2023), 5.
[33] Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. 2016. Summarizing source code using a neural attention model. In *54th Annual Meeting of the Association for Computational Linguistics 2016*. Association for Computational Linguistics, 2073–2083.
[34] Siyuan Jiang, Ameer Armaly, and Collin McMillan. 2017. Automatically generating commit messages from diffs using neural machine translation. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 135–146.
[35] Diederik P Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980* (2014).
[36] Masanari Kondo, Gustavo A Oliva, Zhen Ming Jiang, Ahmed E Hassan, and Osamu Mizuno. 2020. Code cloning in smart contracts: a case study on verified contracts from the Ethereum blockchain platform. *Empirical Software Engineering* 25 (2020), 4617–4675.
[37] Adrian Kuhn, Stéphane Ducasse, and Tudor Gírba. 2007. Semantic clustering: Identifying topics in source code. *Information and software technology* 49, 3 (2007), 230–243.
[38] Xiaoqi Li, Ting Chen, Xiapu Luo, Tao Zhang, Le Yu, and Zhou Xu. 2020. Stan: Towards describing bytecodes of smart contract. In *2020 IEEE 20th International Conference on Software Quality, Reliability and Security (QRS)*. IEEE, 273–284.
[39] Chin-Yew Lin. 2004. Rouge: A package for automatic evaluation of summaries. In *Text summarization branches out*. 74–81.

[40] Yubo Mai, Zhipeng Gao, Xing Hu, Lingfeng Bao, Yu Liu, and JianLing Sun. 2024. Are Human Rules Necessary? Generating Reusable APIs with CoT Reasoning and In-Context Learning. *Proceedings of the ACM on Software Engineering* 1, FSE (2024), 2355–2377.
[41] Paul W McBurney and Collin McMillan. 2015. Automatic source code summarization of context for java methods. *IEEE Transactions on Software Engineering* 42, 2 (2015), 103–119.
[42] Laura Moreno, Jairo Aponte, Giriprasad Sridhara, Andrian Marcus, Lori Pollock, and K Vijay-Shanker. 2013. Automatic generation of natural language summaries for java classes. In *2013 21st International Conference on Program Comprehension (ICPC)*. IEEE, 23–32.
[43] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*. 311–318.
[44] Fangcheng Qiu, Zhipeng Gao, Xin Xia, David Lo, John Grundy, and Xinyu Wang. 2021. Deep just-in-time defect localization. *IEEE Transactions on Software Engineering* 48, 12 (2021), 5068–5086.
[45] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. 2020. Exploring the limits of transfer learning with a unified text-to-text transformer. *The Journal of Machine Learning Research* 21, 1 (2020), 5485–5551.
[46] Stephen Robertson, Hugo Zaragoza, et al. 2009. The probabilistic relevance framework: BM25 and beyond. *Foundations and Trends® in Information Retrieval* 3, 4 (2009), 333–389.
[47] Gerard Salton and Christopher Buckley. 1988. Term-weighting approaches in automatic text retrieval. *Information processing & management* 24, 5 (1988), 513–523.
[48] Selin Sezer, Clemens Eyhoff, Wolfgang Prinz, and Thomas Rose. 2020. Exploiting Smart Contract Bytecode for Classification on Ethereum.. In *PoEM Workshops*. 11–22.
[49] Chaochen Shi, Yong Xiang, Jiangshan Yu, and Longxiang Gao. 2021. Semantic code search for smart contracts. *arXiv preprint arXiv:2111.14139* (2021).
[50] Chaochen Shi, Yong Xiang, Jiangshan Yu, Longxiang Gao, Keshav Sood, and Robin Ram Mohan Doss. 2022. A bytecode-based approach for smart contract classification. In *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 1046–1054.
[51] Chaochen Shi, Yong Xiang, Jiangshan Yu, Keshav Sood, and Longxiang Gao. 2023. Machine translation-based fine-grained comments generation for solidity smart contracts. *Information and Software Technology* 153 (2023), 107065.
[52] Giriprasad Sridhara, Emily Hill, Divya Muppaneni, Lori Pollock, and K Vijay-Shanker. 2010. Towards automatically generating summary comments for java methods. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*. 43–52.
[53] Matt Suiche. 2017. Porosity: A decompiler for blockchain-based smart contracts bytecode. *DEF con* 25, 11 (2017).
[54] Nick Szabo. 1996. Smart contracts: building blocks for digital markets. *EXTROPY: The Journal of Transhumanist Thought,(16)* 18, 2 (1996), 28.
[55] Nick Szabo. 1997. Formalizing and securing relationships on public networks. *First monday* (1997).
[56] Zhaopeng Tu, Zhengdong Lu, Yang Liu, Xiaohua Liu, and Hang Li. 2016. Modeling coverage for neural machine translation. *arXiv preprint arXiv:1601.04811* (2016).
[57] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems* 30 (2017).
[58] Yao Wan, Zhou Zhao, Min Yang, Guandong Xu, Haochao Ying, Jian Wu, and Philip S Yu. 2018. Improving automatic source code summarization via deep reinforcement learning. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. 397–407.
[59] Haoye Wang, Zhipeng Gao, Tingting Bi, John Grundy, Xinyu Wang, Minghui Wu, and Xiaohu Yang. [n. d.]. What Makes a Good TODO Comment? *ACM Transactions on Software Engineering and Methodology* ([n. d.]).
[60] Yilin Wang, Xiangping Chen, Yuan Huang, Hao-Nan Zhu, and Jing Bian. 2022. An empirical study on real bug fixes in smart contracts projects. *arXiv preprint arXiv:2210.11990* (2022).
[61] Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. 2021. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *arXiv preprint arXiv:2109.00859* (2021).
[62] Bolin Wei, Yongmin Li, Ge Li, Xin Xia, David Lo, and Zhi Jin. 2020. Retrieve and Refine: Exemplar-based Neural Comment Generation. In *2020 IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE.
[63] Edmund Wong, Taiyue Liu, and Lin Tan. 2015. Clocom: Mining existing source code for automatic comment generation. In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE, 380–389.
[64] Edmund Wong, Jinqiu Yang, and Lin Tan. 2013. Autocomment: Mining question and answer sites for automatic comment generation. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 562–567.
[65] Gavin Wood et al. 2014. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper* 151, 2014 (2014), 1–32.
[66] Zhipeng Xue, Zhipeng Gao, Shaohua Wang, Xing Hu, Xin Xia, and Shanping Li. 2024. SelfPiCo: Self-Guided Partial Code Execution with LLMs. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 1389–1401.

[67] Dapeng Yan, Zhipeng Gao, and Zhiming Liu. 2023. A Closer Look at Different Difficulty Levels Code Generation Abilities of ChatGPT. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 1887–1898.
[68] Guang Yang, Ke Liu, Xiang Chen, Yanlin Zhou, Chi Yu, and Hao Lin. 2022. CCGIR: Information retrieval-based code comment generation method for smart contracts. *Knowledge-Based Systems* 237 (2022), 107858.
[69] Pengcheng Yin, Graham Neubig, Satoshi Sekine, and Katsuhito Sudoh. 2019. Code Summarization with a Semantic Role-based Attention Mechanism. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics (ACL)*.
[70] Xiao Liang Yu, Omar Al-Bataineh, David Lo, and Abhik Roychoudhury. 2020. Smart contract repair. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 29, 4 (2020), 1–32.
[71] Di Zhu, Feng Yue, Jianmin Pang, Xin Zhou, Wenjie Han, and Fudong Liu. 2022. Bytecode similarity detection of smart contract across optimization options and compiler versions based on triplet network. *Electronics* 11, 4 (2022), 597.