2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE) | 978-1-6654-0337-5/21/$31.00 ©2021 IEEE | DOI: 10.1109/ASE51524.2021.9678552

Automating User Notice Generation for Smart Contract Functions

Xing Hu*‖, Zhipeng Gao†, Xin Xia†¶, David Lo‡ and Xiaohu Yang§
*School of Software Technology, Zhejiang University, Ningbo, China
†Faculty of Information Technology, Monash University, Melbourne, Australia
‡ School of Information Systems, Singapore Management University, Singapore, davidlo@smu.edu.sg
§College of Computer Science and Technology, Zhejiang University, Hangzhou, China
{xinghu,yangxh}@zju.edu.cn, {zhipeng.gao,Xin.Xia}@monash.edu, davidlo@smu.edu.sg

***Abstract*—Smart contracts have obtained much attention and are crucial for automatic financial and business transactions. For end-users who have never seen the source code, they can read the user notice shown in end-user client to understand what a transaction does of a smart contract function. However, due to time constraints or lack of motivation, user notice is often missing during the development of smart contracts. For end-users who lack the information of the user notices, there is no easy way for them to check the code semantics of the smart contracts. Thus, in this paper, we propose a new approach SMARTDOC to generate user notice for smart contract functions automatically. Our tool can help end-users better understand the smart contract and aware of the financial risks, improving the users' confidence on the reliability of the smart contracts. SMARTDOC exploits the Transformer to learn the representation of source code and generates natural language descriptions from the learned representation. We also integrate the Pointer mechanism to copy words from the input source code instead of generating words during the prediction process. We extract 7,878 $\langle function, notice \rangle$ pairs from 54,739 smart contracts written in Solidity. Due to the limited amount of collected smart contract functions (i.e., 7,878 functions), we exploit a transfer learning technique to utilize the learned knowledge to improve the performance of SMARTDOC. The learned knowledge obtained by the pre-training on a corpus of Java code, that has similar characteristics as Solidity code. The experimental results show that our approach can effectively generate user notice given the source code and significantly outperform the state-of-the-art approaches. To investigate human perspectives on our generated user notice, we also conduct a human evaluation and ask participants to score user notice generated by different approaches. Results show that SMARTDOC outperforms baselines from three aspects, *naturalness*, *informativeness*, and *similarity*.**

***Index Terms*—Smart Contract, User Notice Generation, Deep Learning**

I. INTRODUCTION

Recent years have seen an emerging interest in cryptocurrencies (e.g., Bitcoin and Ethereum) on distributed ledgers (a.k.a., Blockchains [1]) from both industry and academia. As one of the largest cryptocurrency platform [2], Ethereum has become a widely used platform to enable financial and business transactions. As the core of Ethereum [3], smart contracts [4][5] are Turing-complete programs and executed on the Ethereum Blockchain. After the deployment, the end-users can interact with a smart contract by sending transactions to its functions. Each transaction consumes a certain amount of "gas" whose price is given in Ethereum cryptocurrency named Ether ($737.15 per unit of Ether as of Dec 2020 [6]). Due to the high stakes of smart contracts and the potential risk of financial loss for users, it is necessary to assist end-users better understand the functionality of smart contracts.

Two groups of people interact with smart contracts: developers and end-users. When developers implement a smart contract, they need to translate financial operations (e.g., transfer) into one or more contract transactions; then the end-users start the transaction which triggers the execution of a function defined within the smart contract. In this study, we argue that the end-users are often non-tech-savvy consumers of the contracts. To assist these users who cannot read the source code, Solidity (one of the most popular programming language for smart contracts) provides a mechanism that can provide notices for end-users. An example of a smart contract function and how it is used by an end-user are illustrated in Figure 1 and Figure 2. Consider Alice is an end-user of smart contracts who knows nothing about programming; when she submits a transaction to the above function with a target address of `0x83***Cc` and "mintedAmount" of 100, then the user notice will be rendered to Alice as: "`Create 100 tokens and send it to 0x83***Cc`". After reading the user notice, Alice can better understand the contract and thus can better make informed decision, i.e., Reject or Confirm the transaction.

Unfortunately, user notices are often lacking for a large number of smart contracts. Even though the official guide line of Solidity recommends that the smart contracts should be annotated with user notice for all public interfaces, this practice is often neglected or ignored by developers during smart contract development. This will make the end-users completely clueless and uninformed, which may discourage the participation of end-users and the usability of the smart contract. Moreover, due to the unalterable feature of the blockchain system, unlike traditional software, user notice can not be added once the smart contract is deployed. Therefore, it is desirable to have a tool that can automatically generate user notices for smart contract developers whenever they forget to

‖Also with PengCheng Laboratory.
¶Corresponding author.

978-1-6654-0337-5/21/$31.00 ©2021 IEEE
DOI 10.1109/ASE51524.2021.00012

```
Source Code:
/// @notice Create `mintedAmount` tokens and send it to `target`
function mintToken(address target, uint256 mintedAmount) onlyOwner public {
    balanceOf[target] += mintedAmount;
    totalSupply += mintedAmount;
    Transfer(0, this, mintedAmount);
    Transfer(this, target, mintedAmount);
}
```

Fig. 1. An example of a smart contract function

do so.

Existing approaches mainly focus on generating comments for common programming languages, e.g., Java and Python. These comments are provided to developers and help them to understand the source code. However, generating comments especially user-oriented comments (i.e., user notices) has not gained much attention yet. Making such a tool for smart contracts is a non-trivial task considering the following challenges: (i) *Dynamic Expressions Mechanism.* The user notice in smart contracts supports dynamic expressions. Different from the general code comments of other programming languages (e.g., Java comments), the Solidity compiler produces the user notice dynamically from the source code. The dynamic expression mechanism requires that certain words in the user notice should be identical with the corresponding tokens in source code. For the example shown in Figure 1, the word "`mintedAmount`" and "`target`" are both copied from the source code. This mechanism causes the user notice to be closely related to the source code. Even though the existing documentation generation approaches have achieved a huge success for general code comments generation (e.g., comments for Java method [7][8][9][10][11]), it is not clear whether they can be successfully applied to user notice generation for smart contracts. How to copy variable names correctly from the smart contracts is still challenging for these deep learning models. (ii) *Data Hungry.* Compared with other popular programming languages (i.e., Java), it is more difficult to collect large-scale datasets for smart contracts. Even though the Ethereum blockchain has accumulated a great number of smart contracts, data hungry problem still exists. That is, only a small proportion of smart contracts have user notices. According to our preliminary study, only 11,409 out of 54,739 smart contracts contain user notices; if we look at the functions, the proportion is even smaller. How to utilize the limited labeled data for generating accurate user notice is challenging for this work.

In this paper, we propose a new approach named SMARTDOC to address the aforementioned challenges. We aim to understand functions in smart contracts and automatically generate user notices (i.e., `@notice`) for functions in a smart contracts. The main idea of our approach is two-folds: (1) while generating user notice, SMARTDOC can predict a word or copy a token from source code. It exploits Transformer [12] equipped with Pointer mechanism to predict user notice. Existing code comment generation approaches usually use Recurrent Neural Network (e.g., LSTM and GRU) to generate code comments. However, these techniques are difficult to capture long-range dependencies between code tokens. In this paper, we exploit the Transformer architecture that can generate user

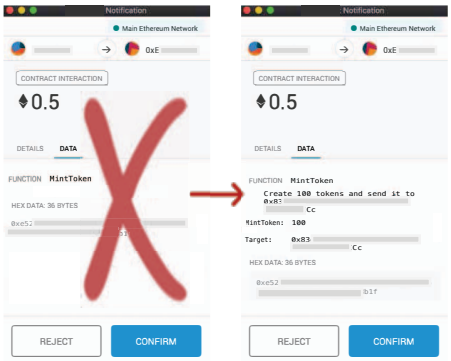


Fig. 2. An example of an end-user submitting a transaction with (on the right) and without (on the left) user notice

notices to reinforce the capability of capturing the long-range dependencies between code tokens. Considering the dynamic expression mechanism, many words in the user notice can be copied from the smart contract functions. Therefore, we integrate the Pointer generator in our approach to overcome the first challenge. The Pointer mechanism can copy a word by pointing tokens in source code. (2) In order to alleviate the limitation of minimal labeled data, we propose to use transfer learning [13] that transfers the knowledge of general comment generation for Java methods into user notice generation for smart contract functions. Solidity and Java languages are somewhat similar in that both are object-oriented and high-level programming languages. Models that have learnt how to convert Java methods into comments can be a good start to the user notice generation.

To evaluate our proposed model, we extract 7,878 $\langle function, notice \rangle$ pairs from 54,739 verified smart contract. The automatic evaluation results show that SMARTDOC achieves the best performance when compared with baselines including attendgru [11], ast-attendgru [11], and Re^{2}Com [10] regarding the he BLEU score and ROUGE-L score. To explore the practitioners' perspective on the generated notice, we also conduct a human evaluation. Each practitioner is asked to evaluate user notice generated by various approaches from three aspects, the *similarity* of generated notice and human-written notice, *naturalness* (grammatical correctness and fluency) of the generated notices, and their *informativeness* (the amount of content carried over from the input code to the generated notices, ignoring fluency of the text). Experiments show that our approach can achieve the best performance when compared to the baseline techniques.

The main contributions of this paper are as follows:

- We are the first to investigate characteristics of smart contract user notices. Our study highlights a problem that has been neglected in the literature but has practical implications.
- We propose a novel approach SMARTDOC for smart contract user notice generation, which aims to help end-users understand smart contracts when they are executed in Blockchain platforms.
- We integrate the Pointer mechanism into Transformer for

better user notice prediction. The approach can generate words or copy tokens from source code. We exploit transfer learning to alleviate the effect of minimal labeled data on training a deep learning model. The experimental results show that our approach outperforms the state-of-the-art techniques.

- We build the first dataset with respect to user notice which contains 7,878 $\langle function, notice \rangle$ pairs. To the best of our knowledge, this is the first dataset of user notices for smart contract functions.

This paper is organized as follows. In Section II, we provide the preliminaries of smart contracts. Section III presents our approach for smart contract user notice generation. Section IV evaluates our approach on actual contracts collected from the Ethereum blockchain. Section V and Section VI illustrate experimental results and practitioners' perspectives on the generated user notice. Section VII discusses our proposed approach. Section IX presents the related works. Section X concludes the paper.

II. Preliminaries

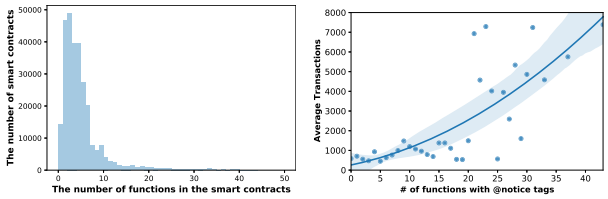
In this section, we present the data hungry issue related to user notices. Then, we show the correlation between user notice and transactions .

A. User notice hungry

Ethereum [14] has attracted increasing attention as a blockchain platform and smart contracts deployed on Ethereum have been applied to many business domains to enable efficient and trustable transactions [15], [16]. When developing smart contracts, Solidity provides a special form of comments, named the Ethereum Natural Language Specification Format (NatSpec), to document contracts and functions [17]. The `@notice` tag is the main NatSpec tag and its audience is end-users. Considering that smart contract end-users are often non-tech-savvy consumers, the user notices can bridge the information gap between smart contract developers and end-users. By interacting with the user notices, the end-users can better assess the financial risks and make better informed decisions. However, according to our preliminary study, only a small proportion of smart contracts include user notices. For example, among the 54,739 contracts that we have collected, only 11,409 of them contain user notices; moreover, only 4% of the functions in smart contracts have user notices.

B. User notice & Transactions

In smart contracts, functions are the executable units of code and can be called by end-users. Intuitively, user notice of functions can help end-users understand the smart contracts, and thus improve the probability of smart contracts' transactions. To verify this conjecture, we collect the transaction information of smart contracts and investigate whether smart contracts with user notice have more transactions. Figure 3 presents the function distribution in smart contracts and the the distribution of the average amount of transactions of smart



(a) Function distribution in Smart Contracts

(b) The average amount of transactions of Smart Contracts with different amount of user notice

Fig. 3. Function Distribution and the Transaction Distribution of Smart Contracts.

```
Contract Name: SMT
Transactions: 58,785
contract Token {
                                                    ...
    /// @notice send `_value` token to `_to` from `msg.sender`
    /// ...
    function transfer(address _to, uint256 _value) public returns (bool success);

    /// @notice send `_value` token to `_to` from `_from` on the condition it is approved by `_from`
    /// ...
    function transferFrom(address _from, address _to, uint256 _value) public returns (bool success);
                                                    ...
}
```

```
Contract Name: LooksCoin
Transactions: 25
contract ERC20 {
                                                    ...
    function transfer(address _to, uint256 _value) public returns (bool success);
    function transferFrom(address _from, address _to, uint256 _value) public returns (bool success);
                                                    ...
}
```

Fig. 4. Motivating Examples

contracts with different amount of user notice. From Figure 3(a), we can observe that almost all smart contracts have less than 40 functions. Thus, we analyze the transactions of them and find that smart contracts with more user notices tend to have more transactions (shown in Figure 3(b)). This is reasonable because detailed user notices can make end-users well aware of the financial risks and improve the users' confidence in the reliability of the smart contracts, therefore end-users prefer to start a transaction through smart contracts with more high-quality user notices.

Figure 4 shows the source code of two smart contracts, i.e., SMT [18] and LooksCoin [19], in which SMT has 58,785 transactions and LooksCoin has 25 transactions. Although these two smart contracts implemented the same functions, such as `transfer` and `transferFrom` in this example, the SMT contract provided adequate user notices while LooksCoin did not make any notice for end-users. The detailed user notice can help end-users better understand of their operations and thus make the smart contract more popular among end-users.

III. Approach

Figure 5 illustrates the overall framework of our approach SmartDoc. It mainly consists of three phases: pre-training, fine-tuning, and application. In the *pre-training* phase, we exploit the Java dataset prepared by Hu et al. [9] to pre-train the model of SmartDoc. Similar to Java, Solidity is also an object-oriented programming language. The two programming languages share similar coding conventions and syntax. Therefore, the knowledge (such as variable naming conventions, sequential information among code tokens) learned from Java can be reused into Solidity. To better exploit the existing

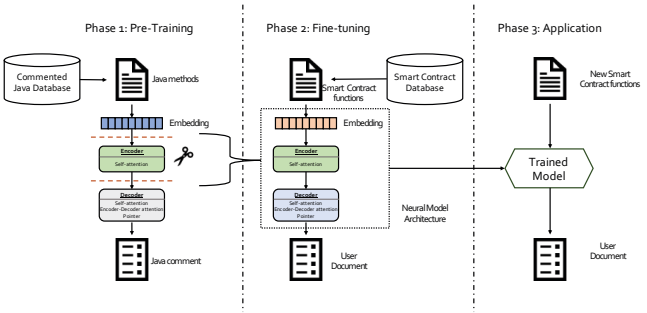


Fig. 5. Overview of our Approach

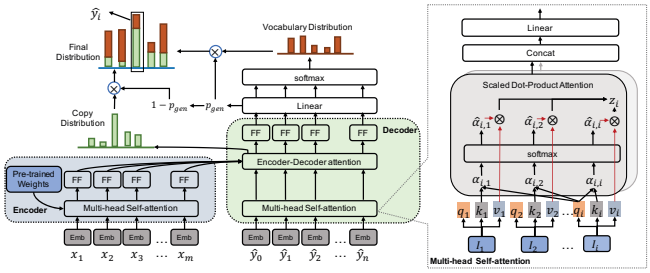


Fig. 6. The structure of our neural network

knowledge of source code, we transfer the pre-trained weights of Java encoder into Solidity encoder.

In the *fine-tuning* phase, we further train the user notice generation model on the corpus of annotated $\langle fun, doc \rangle$ pairs extracted from smart contracts. The encoder is initialized by the learned encoder from the pre-trained model. Except for the source code encoder, parameters of other components are trained from scratch. After training, we can get a trained neural network. Then, given a new function of smart contract, corresponding user notice can be generated by the trained model.

Figure 6 is an overview of the network architecture of our proposed deep learning based model. The architecture of our model follows the Transformer framework [12], [20], [21], which has been successfully adopted in machine translation tasks. The architecture mainly consists of three submodules: (1) Source code encoder. This module aims to represent the source code and exploits the multi-head self-attention to learn the sequential information of the source code. (2) Notice generation decoder. This module aims to generate notice through the self-attention layer and the encoder-decoder attention layer. The encoder-decoder attention layer helps the decoder focus on appropriate places in the input sequence. (3) Pointer generator. The pointer generator [22] is used to copy variables from source code.

We will elaborate on each component in this framework in the following subsections.

A. Encoder

The encoder aims to learn representations for a smart contract function $X = x_1, x_2, ..., x_m$. Each token is embedded into a vector (i.e., $\boldsymbol{X} = (\boldsymbol{x_1}, ..., \boldsymbol{x_m})$) before fed into the encoder. To help SMARTDOC focuses on the important information of the function X, the encoder adopts a multi-head self-attention layer to capture important parts of the input. Then the output of the multi-head self-attention layer is fed into a feed-forward neural network.

The multi-head self-attention layer depicted in Figure 6 exploits scaled dot-product attention to calculate attention weights. Given an input vector $\boldsymbol{I_i} \in \mathbb{R}^d$ (in this paper, $\boldsymbol{I_i}$ represents the embedding of each token), the first step is to create three vectors, i.e., a query vector q_i, a key vector k_i, and a value vector v_i.

Then, we use the query vector q_i of the ith input and the key vector k_j of each word of the input sentence to calculate the attention scores through dot products. The attention score against the ith input is computed as follows:

$$\alpha_{i,j} = \frac{q_i \cdot k_j}{\sqrt{d}} \tag{1}$$

where d is the dimension of q_i and k_j. The score determines how much focus to place on the jth input as we encode the ith input. Then, we get the normalized scores by a $\mathrm{softmax}$ function:

$$\hat{\alpha}_{i,j} = \mathrm{softmax}(\alpha_i) = \frac{\exp(\alpha_{i,j})}{\sum_t \exp(\alpha_{i,t})} \tag{2}$$

To keep the values of the tokens we want to focus on intact, and drown-out irrelevant tokens, we multiply each value vector by the softmax score and sum up the weighted value vectors:

$$z_i = \sum_j \hat{\alpha}_{i,j} v_j \tag{3}$$

For faster processing, the calculation can be done in matrix form, shown as follows:

$$Attention(\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V}) = \mathrm{softmax}(\frac{\boldsymbol{Q}\boldsymbol{K}^T}{\sqrt{d_k}})\boldsymbol{V} \tag{4}$$

In addition, SMARTDOC adopts the multi-head attention with h heads to focus on different channels of the input vectors. The outputs of h heads self-attention are concatenated into one matrix and then are linearly projected by the linear layer: $A = \mathrm{concat}(Attention_i(\boldsymbol{Q}_i, \boldsymbol{K}_i, \boldsymbol{V}_i))W$, where W is the parameter matrix of the linear layer. Then, the outputs of the multi-head self-attention layer are fed into a feed-forward neural network.

B. Decoder

The decoder component mainly consists of two parts, namely, the self-attention layer and the encoder-decoder attention layer. The self-attention layer is similar to that in the encoder component except that it only deals with generated words in the output sequence. Different from the self-attention layer, the encoder-decoder attention layer learns the relationship between the source code and the target user notice. The calculation of attention is similar to self-attention. The Queries matrix $\boldsymbol{Q}$ comes from the output of the self-attention layer and the Key $\boldsymbol{K}$ and Values matrix $\boldsymbol{V}$ from the output of the encoder component. For each step, the decoder outputs a state vector $\boldsymbol{v}$ which can be turned into a word of a sequence.

C. Pointer Generator

As described in Section II, the *Dynamic Expressions Mechanism* of smart contracts user notice causes that words in user notices can be copied from the source code. So we integrate the pointer generator [22] in our approach to solve this problem.

The pointer generator is calculated from the vocabulary distribution P_{vocab}, the copy distribution P_{copy}, and the generation probability p_{gen}.

1) Vocabulary Distribution: According to the Transformer architecture, the vocabulary distribution p_{vocab} is calculated by a softmax layer which follows a linear layer:

$$P_{vocab} = \text{softmax}(\boldsymbol{W} * \boldsymbol{v}) \tag{5}$$

$\boldsymbol{W}$ is a learnable parameter that projects the vector $\boldsymbol{b}$ into a larger vector (i.e., has the same size as the vocabulary size).

2) Copy Distribution: Copy distribution P_{copy} is the probability of copying a word from the input sequence, i.e., source code in this work. It is computed from the attention distribution, namely, the output of the encoder-decoder attention layer in Decoder. The calculation of attention distribution between encoder and decoder is similar to the self-attention layer excepted that the Key vector $\boldsymbol{K}$ is from the outputs of the encoder $\boldsymbol{R} = (\boldsymbol{r_1}, ..., \boldsymbol{r_m})$. The attention distribution $\boldsymbol{\alpha^j}$ between the target words y_j and the source code tokens $w_1, ..., w_m$ is:

$$\boldsymbol{\alpha^j} = \text{softmax}(\frac{\boldsymbol{Q_j K}^T}{\sqrt{d_k}}) \tag{6}$$

Then, the copy distribution P_{copy} is calculated as follows:

$$P_{copy} = \sum_{i:w_i=y_j} \alpha_i^j \tag{7}$$

3) Final Distribution: At last, the model uses a soft switch p_{gen} to choose between generating a word from vocabulary P_{vocab} or copying a token from the input source code P_{copy}. Similar to See et al. [22], the generation probability p_{gen} is for predicting y_j calculated from the concatenation of decoder input $\hat{\boldsymbol{y}}_{\boldsymbol{j}}$, decoder state $\boldsymbol{v_j}$, and the attention distribution $\boldsymbol{\alpha^j}$:

$$p_{gen} = \sigma(\boldsymbol{W_{gen}}[\hat{\boldsymbol{y}}_{\boldsymbol{j-1}}; \boldsymbol{v_j}; \boldsymbol{\alpha^j}] + \boldsymbol{b}) \tag{8}$$

where $\boldsymbol{W_{gen}}$ and $\boldsymbol{b}$ are learnable parameters. σ is the Sigmoid function and $p_{gen} \in [0, 1]$. At last, the final distribution for y_j is:

$$P(y_j) = p_{gen} P_{vocab} + (1 - p_{gen}) P_{copy} \tag{9}$$

D. Transfer Learning

During the training process, deep learning models need large amounts of labeled data. However, the parallel data of smart contract function and notice is limited. To better learn the latent knowledge in smart contracts, we exploit the transfer learning technique to reuse learned knowledge. Transfer learning is an effective technique to alleviate the data hungry issue. Transfer learning goes beyond specific tasks and domains (in this paper, comment generation), and tries to leverage knowledge from pre-trained models and use it to solve target problems (in this paper, user notice generation). Considering the features of smart contract functions, we select comment generation for Java methods as the source task $\mathcal{T}_S$ to learning features of programming language.

The learned knowledge of the Java method, namely, source domain $\mathcal{D}_S$, is then transferred into the target task $\mathcal{T}_T$.

1) Pre-training Procedure: To get a good code representation model for Solidity functions, we first pre-train our model on the Java corpus $\mathcal{D}_S$. This is reasonable due to the following reasons. First, the Java corpus is much larger than the size of the Solidity functions (i.e, 11,409), which has provided sufficient data sets for training a comprehensive model. Second, the Solidity language is similar to the Java language with respect to their grammars and syntax; these similar features learned from Java corpus may also be effective for Solidity functions. To pre-train the model, we exploit the encoder to learn the semantic representation of Java methods and use the decoder to generate Java comments according to the learned representation. The encoder and the decoder are introduced above. The pre-trained encoder contain the knowledge that convert a source code in Java language to the semantic representation. Thus, we can obtain the Java method knowledge $\mathcal{D}_S$ by accessing the pre-trained encoder parameters.

2) Fine-tuning Procedure: When the model is pre-trained, we then fine-tune it on the user notice generation task. The fine-tuning process can quickly adapt the knowledge from the Java pre-trained model to learn the code semantics and structures of Solidity functions. During the fine-tuning procedure, we reused the pre-trained encoder to learn the representation of smart contract functions. The downstream task of generating user notices can be implemented by a decoder which receives user notice representations from the pre-trained encoder. In this way, we can reuse the pre-trained knowledge from the Java programming language.

IV. Evaluation

In this section, we firstly describe the evaluation corpus of the task. We then introduce the baselines to compare and evaluation metrics. Lastly, we explain our experimental settings. The replication package is available[1].

A. Dataset

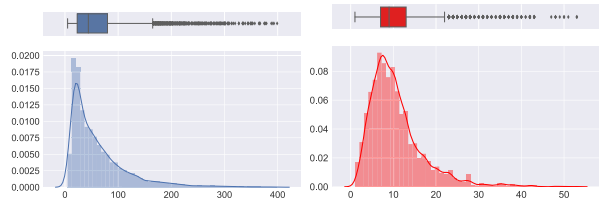
We use the raw smart contract dataset provided by Chen et al. [23], which contains 54,739 verified Solidity files (each file may contain multiple smart contracts) crawled from Etherscan [24]. We describe how we prepared the dataset for user notice generation as follows.

Preprocessing: First, we exploit the Solidity-parser [25] to parse smart contracts and extract functions. We exclude the Constructor functions since they are trivial to generate notice for such functions. As Solidity-parser does not support NatSpec comments extraction, we define regular expressions to

[1]https://github.com/xing-hu/SmartDoc

TABLE I
THE STATISTICS OF OUR COLLECTED SMART CONTRACTS AND FUNCTIONS

Contract	Function	Functions with User notice	Average Function LOC
28,2793	1,296,556	51,567	6.34



(a) Code length distribution (b) User notice length distribution

Fig. 7. Length distribution of the training data

extract NatSpec comments that are tagged with `@notice` for functions. Table I provides the statistics of the preprocessed dataset. We extract 1,296,556 functions from smart contracts and get 51,567 functions with user notices. The average number of Lines of Code (LOC) of smart contract functions is 6.34.

Filtering: Then, we extract the user notices (NatSpec comment labeled `@notice`) from smart contract functions and filtered out non-English samples. Considering that duplicate code has negative impacts on neural networks and introduces bias in evaluation, we remove duplicate functions and user notices. Finally, we get 7,878 $\langle function, notice \rangle$ pairs.

Generating Training/Test sets: We split dataset into training set and test set. We randomly select 1k pairs for testing and the rest for training. Figure 7 illustrates the length distribution of functions and user notice on the training data. We find that more than 96% code snippets have less than 200 tokens and user notice has less than 50 words. In addition, the mode of their lengths are 30 and 10, respectively.

Tokenization: To convert functions into sequential text, we tokenize the source code via Solidity-parser. Then, we tokenize the user notice by Natural Language Toolkit (NLTK) [26]. The vocabulary size of code and notice is 17,844 and 4,692, respectively.

B. Baselines

We compare our model with the following baselines:

1) attendgru: `attendgru` [11] exploits the attentional seq2seq model to generate code comment. It includes an encoder and a decoder that are both gated recurrent unit (GRU) [27]. The encoder aims to learn the representation from the source code and the decoder generates comments from learned representation. They propose to use an attention mechanism to attend words in the output summary sentence to words in the code word representation. During the prediction phase, they use a greedy search algorithm for inference that minimizes the number of experimental variables and computation cost.

2) ast-attendgru: `ast-attendgru` [11] integrates structural information on the basis of `attendgru`. The structural information comes from the abstract syntax tree (AST). In addition to the code encoder, it also contains an encoder to process ASTs. They traverse ASTs into sequences by Structure-based Traversal (SBT) proposed by Hu et al. [8] before fed into neural networks. A separate attention mechanism is used to attend the words to parts of the AST. Then, they concatenate the vectors from each attention mechanism to create a context vector. Finally, they predict the comment one word at a time from the context vector, following what is typical in seq2seq models.

3) Re^{2}Com: `Re`^{2}`Com` [10] is the state-of-the-art code comment generation approach that integrates three kinds of techniques, namely, IR, template, and neural networks. The model consists of two modules: a Retrieve module and a Refine module. In the Retrieve module, Re^{2}Com exploits IR techniques to retrieve the most similar code snippet from a large parallel corpus of code snippets and their corresponding comments, and treat the comment of the similar code snippet as an exemplar. In the Refine module, it applies a novel seq2seq neural network whose encoder takes the given code snippet, the similar code snippet, and the exemplar as input and the decoder generates the token sequence of a comment.

C. Evaluation Metrics

1) BLEU: Following Wei et al. [10], we evaluate different approaches using the metric BLEU [28]. It 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:

$$BLEU = BP \cdot \exp(\sum_{n=1}^{N} w_n \log p_n) \tag{10}$$

where p_n is the precision scores of matched n-grams. In this paper, N is set to 4 that is the same as previous studies. It is widely used in various tasks of automatic software engineering, such as API sequence generation [29], comment generation [8], [9], [10], and commit message generation [30]. We reuse the evaluation script provided by Wei et al. [10] to compute the BLEU scores.

2) ROUGE-L: ROUGE-L [31] is the other widely used metric that takes into account sentence level structure similarity naturally and identifies longest co-occurring in sequence n-grams automatically. For two sentences X (with length m) and Y (with length n), in which X is a reference and Y is a candidate generated sentence. ROUGE-L first calculates the precision and recall the longest common subsequence of them, i.e.,

$$P_{lcs} = \frac{LCS(X,Y)}{n} \tag{11}$$

$$R_{lcs} = \frac{LCS(X,Y)}{m} \tag{12}$$

TABLE II
COMPARISONS OF SMARTDOC WITH EACH BASELINE IN TERMS OF BLEU AND ROUGE-L (*P<0.05)

Approaches	BLEU	B1	B2	B3	B4	R-L
attendgru	29.01	37.39	28.52	26.41	25.15	38.48
ast-attendgru	26.01	33.75	25.71	23.61	22.34	34.51
Re^{2}Com	29.37	41.39	28.99	25.77	24.07	34.55
SMARTDOC	**47.39***	**56.51***	**46.78***	**44.27***	**43.08***	**51.86***

Then, the final score is calculated according to P_{lcs} and R_{lcs}:

$$F_{lcs} = \frac{(1+\beta^2)R_{lcs}P_{lcs}}{R_{lcs}+\beta^2 P_{lcs}} \quad (13)$$

We follow the default value of β provided in [31] and set it as 1.

D. Training Details

We implement SMARTDOC on top of TensorFlow [32]. Both token embeddings and hidden size are set to 256 dimensions. In addition, the number of attention heads and blocks are set to 4 and 3, respectively. All parameters are optimized using Adam [33] with the initial learning rate of 0.0005. Following Vaswani et al. [12], 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 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.

V. RESULTS

To gain a deeper understanding of the performance of our approach, we conduct analysis on our evaluation results in this section. Specifically, we focus on three research questions:

- How effective is our SMARTDOC for generating user notice given smart contract functions?
- How effective is each component of SMARTDOC?
- How efficient is SMARTDOC?

A. RQ1: SMARTDOC Overall Effectiveness (vs. Baselines)

In this RQ, we want to investigate how effective our approach is and how much performance improvement our approach can achieve over the baselines.

1) Experimental Setup: We apply our approach and the baseline methods (i.e., `attendgru`, `ast-attendgru`, and `Re$^2$Com`) on the collected dataset, and compare their performance in terms of BLEU and ROUGE-L. To check whether the performance differences between SMARTDOC and baseline approaches are significant, we run Wilcoxon signed-rank tests [34] at the confidence level of 95%. For each approach, we collect 1,000 scores (one for each case) considering every evaluation metric. Then, we conduct the Wilcoxon signed-rank test for each pair of competing approaches considering these scores.

TABLE III
EFFECTIVENESS OF EACH INCOMPLETE VARIANT OF OUR APPROACH IN TERMS OF BLEU AND ROUGE-L.(*P<0.05)

Approaches	BLEU	B1	B2	B3	B4	R-L
Transformer	43.52	52.02	43.2	40.56	39.34	49.91
Transformer+P	45.12	53.23	44.59	42.34	41.25	48.55
SMARTDOC	**47.39***	**56.51***	**46.78***	**44.27***	**43.08***	**51.86***

2) Results: Table II shows the BLEU and ROUGE-L scores for our approach SMARTDOC and the baseline techniques. We can observe that `ast-attendgru` has the worst performance and our approach SMARTDOC outperforms all baseline models significantly. Compared to baselines, the improvements of our proposed approach SMARTDOC are more than 60% and 35% in terms of BLEU and ROUGE-L, respectively. All the p-values are substantially smaller than 0.05, which means our approach significantly improves over the baseline approaches.

B. RQ2: Ablation Analysis

In this paper, we use the Transformer to model the source code of functions in smart contracts. Then, we exploit the pointer mechanism to copy words from the source code during the user notice generation process. To alleviate the limitation of minimal labeled data, we also leverage the transfer learning to utilize the pre-trained knowledge before predicting the user notice. We want to investigate the impacts of these components on the performance of our approach.

1) Experimental Setup: To illustrate the importance of each component, we compare our approach with two of its incomplete variants:

- `Transformer` removes both the pointer mechanism and the pre-trained knowledge.
- `Transformer+P` is transformer with pointer and only removes the pre-trained knowledge.

We can observe the effectiveness of pointer mechanism and transfer learning technique by comparing SMARTDOC and `Transformer`. Then, we compare SMARTDOC and `Transformer+P` to measure the improvements from the pre-trained knowledge. Similar to RQ1, BLEU and ROUGE-L are used to evaluate our approach and the two variants. The Wilcoxon signed-rank test is also computed.

2) Results: The effectiveness of the two variants are demonstrated in Table III. We can observe that our approach SMARTDOC outperforms the other two variants on each metric. Compared to experimental results of baselines shown in Table II, both `Transformer` and `Transformer+P` outperforms baselines. Specifically, the Transformer boosts the performance by a large margin (more than 40% in terms of BLEU and 30% in terms of ROUGE-L) on user notice generation compared to baselines that are based on recurrent neural network. The integration of Pointer mechanism improves the performance further (with improvement of about 4% in terms of BLEU) – as compared to `Transformer`.

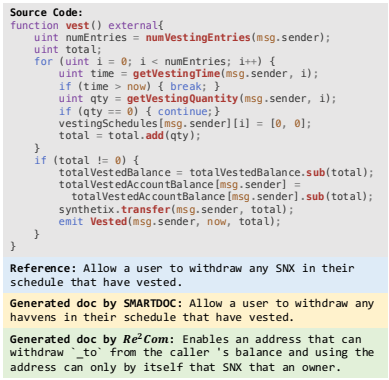


Fig. 8. A test example with copied words

It helps copy words from smart contract instead of generating words during the prediction phase. However, the ROUGE-L of `Transformer+P` is slightly lower than that of `Transformer`. The reason of the slight drop is that the Pointer usually copies one word instead of N-grams, thus, the longest co-occurring n-grams are lower than sequences generated by the `Transformer`. Compared to the `Transformer+P`, SMARTDOC exploits the transfer learning technique to alleviate the limitation of the amount of the dataset. We can observe that it improves about 4% and 7% in terms of BLEU and ROUGE-L, respectively. In addition, the P-value of the improvements is less than 0.05 that indicates out approach SMARTDOC significantly outperforms variants.

3) Effectiveness of Pointer mechanism: In this paper, we integrate Pointer mechanism to copy words from source code during the user notice prediction. It further improves the accuracy of generated notice. To figure out which tokens will be copied into the notice, we manually inspect the test results of the predictions. We find that our approach usually copies tokens with natural features, such as function names and parameter names. Figure 8 shows an example whose generated notice contains words copied from source code. Word "vested" is directly copied from the function name in the emit statement. In addition, some words are copied from the sub-words in specific parameter names. For example, "schedule" is derived from parameter "vestingSchedules".

C. RQ3: Time Costs of our Approach

Neural network models need to be trained before being adapted to generate user notice for smart contracts. The training process is conducted offline and can be used to make prediction online. In this research question, we want to investigate the training time cost and the test time cost of our approach to better understand the practicality of our approach SMARTDOC.

1) Experimental Setting: To measure the time complexity of our approach and other baselines, we record the start time and the end time of their training process and the test process. For fair comparison, all models are trained on the same machine containing a NVIDIA GeForce RTX 2080Ti GPU with 10 GB memory.

TABLE IV
TIME COSTS OF DIFFERENT APPROACHES

Approaches	Train	Test	Test One	Params
attendgru	0.5h	3.04s	0.004s	16.9 M
ast-attendgru	0.9h	4.67s	0.005s	17.4 M
Re^{2}Com	8.2h	16s	0.02s	18.1 M
SMARTDOC	0.59h	72s	0.07s	25.0M

2) Results: Table IV illustrates the time costs of our approach and baselines. Compared to baselines, our approach SMARTDOC has the most parameters with about 25M trainable parameters. Re^{2}Com incurs the highest cost (about 8.2 hours) to train the model well. During the test phase, approach attendgru is the most efficient approach; on average, it takes 0.004 second to recommend user notice. However, the quality of user notice generated by it is limited. Our approach, SMARTDOC takes about 0.59 hour to be trained well and takes about 70ms to recommend a user notice. The experimental results demonstrate that our approach is efficient for practical uses.

VI. HUMAN EVALUATION

Although automatic metrics, such as BLEU and ROUGE-L, can evaluate the gap between the generated user documentation and reference texts written by humans, it can not reflect the human perceptions on the generated user documentation. We follow Wei et al. [10] to conduct human evaluation. The human evaluation measures three aspects, including the *Similarity* of generated user documentation and references, *Naturalness* (grammaticality and fluency of the generated user documentation), and *Informativeness* (the amount of content carried over from the input code to the generated user documentation, ignoring fluency of the text). The scores range from 0 to 4 (the higher the better). We invite 10 volunteers with 1-3 years of blockchain or smart contract experience and have good English proficiency for 30 minutes each to evaluate the generated user documentation in the form of a questionnaire. We randomly select 100 smart contract functions and provide references (human-written user documentation) for each function. In addition, we also provide four machine-generated user documentation that is generated by `attendgru`, `ast-attendgru`, `Re2Com`, and SMARTDOC , respectively. Each participant is asked to score each sample considering *similarity*, *naturalness*, and *informativeness* aspects. All these scores are integers, ranging from 0 to 4. During the annotation, participants are allowed to search the Internet for related information and unfamiliar concepts. Participants do not know which approach produces which texts.

Table V presents the human evaluation results of our approach and baselines. SMARTDOC outperforms other techniques in three aspects, especially in *Informativeness* (improvements more than 30%) and *Similarity* (improvements

TABLE V
THE RESULTS (STANDARD DEVIATION σ IN PARENTHESES) OF HUMAN EVALUATION (*P<0.05)

Approaches	**Informativeness**	**Naturalness**	**Similarity**
attendgru	1.80 (1.44)	3.03 (1.04)	1.71 (1.44)
ast-attendgru	1.85 (1.43)	3.06 (1.08)	1.74 (1.35)
Re^{2}Com	2.07 (1.37)	3.24 (0.86)	2.06 (1.30)
SMARTDOC	**2.69*** (1.31)	**3.45*** (0.73)	**2.63*** (1.34)

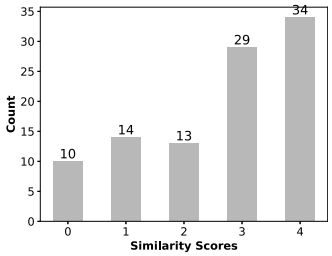


Fig. 9. The count of *similarity* scores of the generated user notices by SMARTDOC compared with human-written user notices.

more than 28%). *Naturalness* scores of the four approaches (all more than 3) are much higher than *Informativeness* and *Similarity* scores, indicating that almost all generated user documentation is grammatical and fluent. Similar to Wei et al [10], the difference in standard deviation of the four methods is also very small, indicating that their scores are about the same degree of concentration. All the p-values are substantially smaller than 0.05, which shows the improvements of our proposed model are statistically significant.

Figure 9 shows the count of similarity scores of user notices generated by SMARTDOC. We can find that 34% generated user notices are almost the same as human-written ones (i.e., generated user notices with score 4). 29% generated user notices are scored as 3 that means 29% of them share many similar words and express similar meanings. In other words, 63% of user notices generated by SMARTDOC are similar to human-written ones and can be added to the source code.

VII. DISCUSSION

In this section, we discuss the performance of the generated user notice and analyze the effectiveness of our approach.

A. Cross-Smart contract validation

Different from mature smart contracts with many transactions, new smart contracts often lack sufficient training data. Thus it is difficult to directly apply our approach to generate user notice for a new smart contract. This problem may be overcome through the cross-project prediction, which uses the data collected from mature smart contracts to train a model, and applies the trained model to make predictions for new smart contracts. We want to investigate whether our approach is still effective for the cross-project setting.

TABLE VI
THE AVERAGE BLEU AND ROUGE-L SCORES OF SMARTDOC AND BASELINES IN CROSS-PROJECT SETTING.

Approaches	BLEU	B1	B2	B3	B4	R-L
attendgru	30.83	38.62	30.34	28.4	27.15	39.67
ast-attendgru	28.5	35.99	28.03	26.18	25	36.32
Re^{2}Com	30.63	40.15	30.13	27.68	26.31	36.10
SMARTDOC	**43.44**	**50.26**	**42.81**	**41.11**	**40.24**	**47.79**

1) Experimental Setting: We conduct a cross-project validation experiment for our collect smart contracts. This process repeats 5 times and we train various models on functions from 80% smart contracts and test these models on functions from the rest 20% smart contracts each time. BLEU scores and ROUGE-L are used to measure the effectiveness of cross-project predictions.

2) Results: Table VI illustrates the cross-project validation for user notice generation. It shows the average scores from the 5 times experiments. We can observe that our approach SMARTDOC still outperforms baselines significantly. Compared to results in RQ1, the performance of baselines (i.e., `attendgru`, `ast-attendgru`, and SMARTDOC) improves whereas SMARTDOC decreases in cross-project setting. The smart contracts come from different domains, which introduces the decrease of performance of our approach SMARTDOC. For baselines, they have more samples to train the models and can improve the performance.

B. Investigating the effectiveness of Transfer Learning

To address the user notice hungry problem, we employ the Transfer Learning technique to transfer the knowledge of comment generation for Java into user document generation for smart contracts. We would like to investigate whether the Transfer Learning is effective for general neural models. To do this, we first analyze the convergence time of SMARTDOC with/without transfer learning, after that, we further explore the performance of baseline models after adding Transfer Learning techniques.

1) Experimental Settings.: First, we record the results of SMARTDOC and `Transformer+P` for every 2,000 steps (about 10 epochs) during the training process, and compare their convergence. Then, we equip each baseline with transfer learning technique to explore their performance with transferred knowledge. Similar to SMARTDOC , we first pre-train various models with Java dataset. As Leclair et al [11] provide trained models in their replication, we reuse their trained models before training on the smart contract dataset. Then, we reuse the weights of source code encoder and fine tune them on the smart contract dataset. Finally, we evaluate them in terms of BLEU scores and ROUGE-L.

2) Results.: Figure 10 presents the convergence of SMARTDOC and `Transformer+P`, in which SMARTDOC uses transfer learning whereas `Transformer+P` not. We can observe that SMARTDOC outperforms `Transformer+P` about 15% in terms of ROUGE-L after training 2,000 steps. In

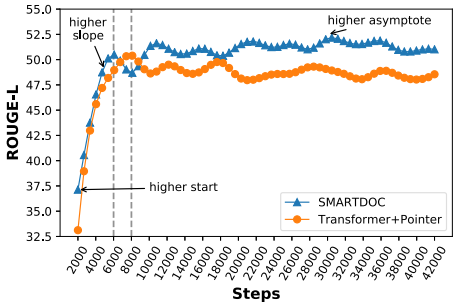


Fig. 10. The convergence of SMARTDOC (with transfer learning technique) and Transformer+P

TABLE VII
COMPARISONS OF SMARTDOC WITH EACH BASELINE IN TERMS OF BLEU AND ROUGE-L WHEN INTEGRATED TRANSFER LEARNING.

Approaches	BLEU	B1	B2	B3	B4	R-L
attendgru + T	29.05	38.92	28.58	26.07	24.57	40.15
ast-attendgru + T	29.01	38.43	28.66	26.13	24.6	38.92
Re^{2}Com + T	25.66	39.59	25.46	21.62	19.89	31.32
SMARTDOC	**47.39**	**56.51**	**46.78**	**44.27**	**43.08**	**51.86**

addition, we find that the convergence of Transformer+P is slow by 2,000 steps compared to SMARTDOC. In addition, we can observe that the SMARTDOC convergence curve has higher start, higher slope, and higher asymptote. The higher start means that our model has the better initial ability (before refining the model) to generate user notice. The higher slope indicates that rate of improvement of our model during training is steeper. Finally, the higher asymptote shows a better converged skill of SMARTDOC.

Table VII shows the results of all models integrated the transfer learning technique. We can find that attendgru and ast-attendgru perform better when integrated transfer learning and ast-attendgru has biggest improvement (11.8% in terms of BLEU) among them. However, Re^{2}Com performs worse when integrated transfer learning technique. The effectiveness of Re^{2}Com comes from two parts, template notice and the source code. As the pre-trained knowledge of source code is affected by the retrieved template notice, thus lead to worse performance of Re^{2}Com. In summary, the transfer learning helps models leverage existing knowledge when training on a new task. It can alleviate the limit of the amount of dataset and improve performance of neural models.

C. Investigating the User Notice with Low Scores

We are also curious about why our approach fail to generate accurate user notice. The reasons are demonstrated in the following subsections.

1) Abbreviations: Different from other software, smart contracts development is related to currencies and finance. Thus, there are many abbreviations in human-written notice. For example, Figure 11 illustrates an example human-written notice containing many abbreviations, including "DOL", "VAULT", and "ETH". However, in the user notice generated by SMARTDOC, these abbreviations are replaced by the full version of the word or phrase. "DOL" is the symbol for "metadollar" [35] that is generated by SMARTDOC. Both "ether" and "ETH" represent the native cryptocurrency token of the Ethereum platform. Although these words are regarded as error tokens during evaluation, they describe the same meaning.

Source Code:
```
function buy() payable public {
    require(!frozenAccount[msg.sender]);
    require(msg.value > 0);
    buyToken();
}
```
Reference: Buy DOL from VAULT by sending ETH

Generated doc by SMARTDOC: Buy metadollars from contract by sending ether

Generated doc by *Re^{2}Com*: Used in transport, challenge and train, to get the genes of a specific hero, a claim a hero if didn't have any.

Fig. 11. A test example whose reference contains abbreviations

Source Code:
```
function approveAndCall(address _spender, uint256 _value,
                        bytes _extraData) returns (bool success) {
    mimonedarecipiente spender = mimonedarecipiente(_spender);
    if (approve(_spender, _value)) {
        spender.receiveApproval(msg.sender,_value,this,_extraData);
        return true;
    }
}
```
Reference: Allows `_spender` to spend no more than `_value` tokens in your behalf, and then ping the contract about it

Generated doc by SMARTDOC: Allows `_spender` to send funds on your behalf, and then ping the contract about it

Generated doc by *Re^{2}Com*: `msg.sender` approves `_spender` to send `_value` tokens on its behalf, and then a function is triggered in the contract that is being approved, `_spender`.

Fig. 12. A test example with missed details

2) Less detailed information: Some details are missed in the generated notice. For example, Figure 12 illustrates an example whose details are missing in the generated user notice. We can find that information "no more than '_value' tokens" is missing in the notice generated by SMARTDOC. This missed information causes the low score during evaluation. In addition, we find that the missed information is hard to be learned from the given input source code. It is usually implied in API invocations, such as "no more than '_value' tokens" can be obtained in the invocation of API "approve".

D. User notice generation VS. Comment generation

Problem Difference: The audience of code comments is developers who can understand the source code, and code comments often contain many details such as the implementation details. Considering the audiences of smart contracts include developers and end-users, comments of smart contracts are divided into two types, i.e., the comment for developers (tagged with @dev) and the comment for end-users (tagged with @notice). Comments for developers (tagged with @dev) are similar with general code comments, while comments for end-users (tagged with @notice) are a special type of comment, whose audience is contract end users who are unable to read the source code. We refer to this kind of special comment as user notice in this study.

Technique Difference: Compared to code comments, user notices support dynamic expression mechanisms that usually copy variable names from source code. These variables

will dynamically be replaced by corresponding values when end users interact with the contract. Compared to comment generation, user notice generation is more dependent on the copy mechanism. In addition, the dataset of code comment generation is large enough to train a model well. However, the dataset for user notice generation is limited, thus, we need to exploit pre-trained models to better avoid overfitting on small data.

VIII. Threats to Validity

We have identified the following threats to validity among our study:

Internal Validity In this paper, we exploit Java dataset to pre-train the models and then use the learned knowledge by using the trained parameters. Pre-trained knowledge learned from different programming languages may bias the effectiveness of our approach on user notice generation. We will try to employ datasets in different programming languages (e.g., Python and Javascript) to pre-train the code knowledge in the future.

Data Validity We use the smart contracts provided by Chen et al. [23] that contains 54,739 smart contracts. However, because a very large number of smart contracts are copied from other smart contracts [36], [37], [38], [39], the number of collected $\langle function, notice \rangle$ pairs is limited after deduplication. The selected contracts may not be sufficiently diverse or representative of all contracts. To mitigate the threat, we conduct experiments on cross-project setting. The trained models are applied to generate user notice for new smart contracts. We believe that our approach is effective for new smart contracts if it performs well on cross-project validity.

External Validity We validate our approach by comparing the generated user notice and human-written user notice. We assume that human-written user notice is correct. However, human-written user notice may also not correct sometimes and we can not ensure the quality of it. For example, some human-written user notice may be outdated during the development. Therefore, the results may be biased and incomprehensive.

IX. Related work

Code comment generation is the most relevant task which aims to generate natural descriptions for code snippets. Manually-crafted templates [40], [41], [42], IR techniques [43], [7], [44], [45], and neural models [11], [8], [10], [46], [47] are widely used in automatic comment generation.

Approaches based on manually-crafted templates usually leverage stereotype identification techniques to generate comments for code snippets. Sridhara et al. [41] 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. Mcburney et al. [42] exploit SWUM to extract keywords from Java methods and use PageRank to select the most important methods from a given context.

Information Retrieval (IR) techniques are widely used in comment generation task. Generally, these approaches first retrieve similar code snippets with comments and take their comments as the output. Latent Semantic Indexing (LSI), Vector Space Model (VSM), and Latent Dirichlet Allocation (LDA) are widely used in comment generation. Kuhn et al. [43] propose to use the Latent Semantic Indexing (LSI) technique to extract topics that reflect the intention of source code. Haiduc et al. [7] exploit two IR techniques, Vector Space Model (VSM) and LSI, to analyze methods and classes in Java projects and generate short descriptions for them. Different from these works, Wong et al. [44], [45] exploit clone detection techniques to retrieve similar code snippets and use corresponding comments for comment generation.

In recent years, considerable attention has been paid to neural networks on comment generation. Iyer et al. [48] 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. Soon after, Hu et al. [8] propose to integrate structural information while generating comments for Java methods. They propose a new approach to traverse an AST into a sequence and encode the sequence by an LSTM. Some studies [49], [10] combine the IR-based techniques and deep-learning-based techniques to generate code comments. Wei et al. [10] propose an approach that takes the advantages of manually-crafted templates, IR, and neural networks techniques. It first retrieves a similar code snippet from the training set and uses its comment as the exemplar to guide the neural model for comment generation.

X. Conclusion and Future Work

In this paper, we propose a new approach, SmartDoc, based on Transformer, Pointer mechanism, and transfer learning technique for smart contract user notice generation. We have evaluated our approach with 1,000 pairs of smart contract functions and their user notice. Experimental results show that it can effectively generate user notice for smart contracts and outperforms the state-of-the-art approaches significantly. In addition, we conduct human evaluation to investigate the human perspectives on the generated user notice. The results show that our approach can generate natural and informative user notice, and the generated user notice is much more similar to the reference text than baselines.

XI. Acknowledgments

This research was partially supported by the National Science Foundation of China (No. U20A20173), Key Research and Development Program of Zhejiang Province (No.2021C01014), and the National Research Foundation, Singapore under its Industry Alignment Fund – Prepositioning (IAF-PP) Funding Initiative. Any opinions, findings, and conclusions, or recommendations expressed in this material are those of the author(s) and do not reflect the views of the National Research Foundation, Singapore.

REFERENCES

[1] N. Radziwill, "Blockchain revolution: How the technology behind bitcoin is changing money, business, and the world," *The Quality Management Journal*, vol. 25, no. 1, pp. 64–65, 2018.

[2] W. Chen, M. Ma, Y. Ye, Z. Zheng, and Y. Zhou, "Iot service based on jointcloud blockchain: The case study of smart traveling," in *2018 IEEE Symposium on service-oriented system engineering (SOSE)*. IEEE, 2018, pp. 216–221.

[3] T. Chen, Z. Li, Y. Zhu, J. Chen, X. Luo, J. C.-S. Lui, X. Lin, and X. Zhang, "Understanding ethereum via graph analysis," *ACM Transactions on Internet Technology (TOIT)*, vol. 20, no. 2, pp. 1–32, 2020.

[4] N. Szabo, "Smart contracts," 1994.

[5] M. Alharby and A. Van Moorsel, "Blockchain-based smart contracts: A systematic mapping study," *arXiv preprint arXiv:1710.06372*, 2017.

[6] https://etherscan.io/chart/etherprice.

[7] S. Haiduc, J. Aponte, L. Moreno, and A. Marcus, "On the use of automated text summarization techniques for summarizing source code," in *2010 17th Working Conference on Reverse Engineering*. IEEE, 2010, pp. 35–44.

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

[9] ——, "Deep code comment generation with hybrid lexical and syntactical information," *Empirical Software Engineering*, vol. 25, no. 3, pp. 2179–2217, 2020.

[10] B. Wei, Y. Li, G. Li, X. Xia, D. Lo, and Z. Jin, "Retrieve and refine: Exemplar-based neural comment generation," in *2020 IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2020.

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

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

[13] S. J. Pan and Q. Yang, "A survey on transfer learning," *IEEE Transactions on knowledge and data engineering*, vol. 22, no. 10, pp. 1345–1359, 2009.

[14] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making smart contracts smarter," in *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, 2016, pp. 254–269.

[15] Z. Wan, X. Xia, and A. Hassan, "What do programmers discuss about blockchain? a case study on the use of balanced lda and the reference architecture of a domain to capture online discussions about blockchain platforms across stack exchange communities," *IEEE Trans Softw Eng*, vol. 2019, pp. 1–1, 2019.

[16] Z. Wan, X. Xia, D. Lo, J. Chen, X. Luo, and X. Yang, "Smart contract security: a practitioners' perspective," *arXiv preprint arXiv:2102.10963*, 2021.

[17] https://solidity.readthedocs.io/en/v0.7.0/natspec-format.html.

[18] https://etherscan.io/address/0x55F93985431Fc9304077687a35A1BA103dC1e081#code.

[19] https://etherscan.io/address/0x8b5fe260f2b7bdd03efc833f78557a349600ae46#code.

[20] Q. Wang, B. Li, T. Xiao, J. Zhu, C. Li, D. F. Wong, and L. S. Chao, "Learning deep transformer models for machine translation," in *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, 2019, pp. 1810–1822.

[21] Y. You, W. Jia, T. Liu, and W. Yang, "Improving abstractive document summarization with salient information modeling," in *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, 2019, pp. 2132–2141.

[22] A. See, P. J. Liu, and C. D. Manning, "Get to the point: Summarization with pointer-generator networks," in *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 2017, pp. 1073–1083.

[23] J. Chen, X. Xia, D. Lo, and J. Grundy, "Why do smart contracts self-destruct? investigating the selfdestruct function on ethereum," *arXiv preprint arXiv:2005.07908*, 2020.

[24] https://etherscan.io/.

[25] https://pypi.org/project/solidity-parser/.

[26] E. Loper and S. Bird, "Nltk: the natural language toolkit," in *Proceedings of the ACL-02 Workshop on Effective tools and methodologies for teaching natural language processing and computational linguistics-Volume 1*, 2002, pp. 63–70.

[27] J. Chung, C. Gulcehre, K. Cho, and Y. Bengio, "Empirical evaluation of gated recurrent neural networks on sequence modeling," *arXiv preprint arXiv:1412.3555*, 2014.

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

[29] X. Gu, H. Zhang, D. Zhang, and S. Kim, "Deep api learning," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2016, pp. 631–642.

[30] S. Jiang, A. Armaly, and C. McMillan, "Automatically generating commit messages from diffs using neural machine translation," in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2017, pp. 135–146.

[31] C.-Y. Lin, "ROUGE: A package for automatic evaluation of summaries," in *Text Summarization Branches Out*. Barcelona, Spain: Association for Computational Linguistics, Jul. 2004, pp. 74–81. [Online]. Available: https://www.aclweb.org/anthology/W04-1013

[32] https://www.tensorflow.org/.

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

[34] F. Wilcoxon, "Individual comparisons by ranking methods," in *Breakthroughs in statistics*. Springer, 1992, pp. 196–202.

[35] https://etherscan.io/address/0x4414a8c55fcbd70a6957e3cb9561e60f0b4e742d.

[36] J. Chen, X. Xia, D. Lo, J. Grundy, and X. Yang, "Maintaining smart contracts on ethereum: Issues, techniques, and future challenges," *arXiv preprint arXiv:2007.00286*, 2020.

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

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

[39] Z. Gao, V. Jayasundara, L. Jiang, X. Xia, D. Lo, and J. Grundy, "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, 2019, pp. 394–397.

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

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

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

[43] A. Kuhn, S. Ducasse, and T. Gírba, "Semantic clustering: Identifying topics in source code," *Information and software technology*, vol. 49, no. 3, pp. 230–243, 2007.

[44] E. Wong, J. Yang, and L. Tan, "Autocomment: Mining question and answer sites for automatic comment generation," in *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2013, pp. 562–567.

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

[46] U. Alon, S. Brody, O. Levy, and E. Yahav, "code2seq: Generating sequences from structured representations of code," in *International Conference on Learning Representations*, 2018.

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

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

[49] J. Zhang, X. Wang, H. Zhang, H. Sun, and X. Liu, "Retrieval-based neural source code summarization," in *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. IEEE, 2020, pp. 1385–1397.