Self-Supervised Learning of Smart Contract Representations

Shouliang Yang, Xiaodong Gu, Beijun Shen*
School of Software, Shanghai Jiao Tong University
Shanghai, China
{ysl0108,xiaodong.gu,bjshen}@sjtu.edu.cn

ABSTRACT

Learning smart contract representations can greatly facilitate the development of smart contracts in many tasks such as bug detection and clone detection. Existing approaches for learning program representations are difficult to apply to smart contracts which have insufficient data and significant homogenization. To overcome these challenges, in this paper, we propose SRCL, a novel, self-supervised approach for learning smart contract representations. Unlike existing supervised methods, which are tied on task-specific data labels, SRCL leverages large-scale unlabeled data by self-supervised learning of both local and global information of smart contracts. It automatically extracts structural sequences from abstract syntax trees (ASTs). Then, two discriminators are designed to guide the Transformer encoder to learn local and global semantic features of smart contracts. We evaluate SRCL on a dataset of 75,006 smart contracts collected from Etherscan. Experimental results show that SRCL considerably outperforms the state-of-the-art code representation models on three downstream tasks.

KEYWORDS

Smart Contract, Self-supervised Learning, Code Representation Learning, Data Augmentation

ACM Reference Format:
Shouliang Yang, Xiaodong Gu, Beijun Shen. 2022. Self-Supervised Learning of Smart Contract Representations. In *30th International Conference on Program Comprehension (ICPC '22), May 16–17, 2022, Virtual Event, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3524610.3527894

1 INTRODUCTION

Smart contracts, the universal and vital programs that are deployed on blockchains, have gained increasing attention with the rapid development of blockchains. For example, more than 10 million smart contracts have been deployed on the Ethereum Mainnet [18]. A smart contract is an event-driven, state-based program that is written in high level languages such as Solidity[1]. Smart contracts have been widely used in many business domains to enable efficient and trustful transactions [38, 42, 46].

*Beijun Shen is the corresponding author.
[1]https://solidity.readthedocs.io/en/v0.6.0/

Unlike general programs, the development of smart contracts requires special effort due to their unique characteristics. First, smart contracts are more bug intolerant compared with general programs. "Code is law", a smart contract can not be modified once it has been released. This is because transactions of a smart contract always involve cryptocurrencies which are worthy of millions of dollars (*e.g.* The DAO). A bug in a smart contract may lead to a substantial loss. Therefore, ensuring the correctness of contracts before releasing is critical. This requires us to reuse experience of developed contracts in the past when developing new contracts. Program mining for smart contracts such as summarization [55], checking [14], and code search [45] can greatly facilitate the development and maintenance of smart contracts.

Learning smart contract representations, namely, converting a smart contract into a continuous, high-dimensional vector, acts as the core process in program mining [3]. Like common programming languages, smart contracts are composed of lexical tokens and parsing trees. Such discrete and structural data can hardly be understood by machine learning models [20]. For example, there can be special relationships between smart contracts. However, capturing semantic similarities between contracts is nontrivial by text matching due to the numerous variants in terms of variables and structures. Hence, it is strongly demanded to convert smart contracts into dense, continuous vectors that reflect their semantics.

Although learning program representations has been well studied, learning smart contract representations faces many challenges. Compared to general programming languages, there is often a scarce availability of large and labeled datasets for training deep code representation models. It is costly and laborious to build a large scale and high-quality dataset with human labeling. For example, an image segmentation dataset containing 10k+ high-quality samples could cost up to a million-dollar [43].

Second, and more critically, smart contracts are highly homogeneous and redundant. In order to gain trusts from users, authors of smart contracts often publicly open source their contracts. As a result, developers tend to clone existing contracts rather than taking the risk of coding from scratch. According to the statistics by Chen et al. [8], about 26% contract code blocks are cloned at an average of 14.6 times. Only 20.8% of the studied contracts are completely original [27]. Such data redundancy can cause machine learning models to be poorly fitted especially on scarce training samples. This causes further issues such as clone related bugs and plagiarism.

To address the aforementioned challenges, we propose SRCL, a novel approach for learning smart contract representations. SRCL is based on self-supervised learning. That is, it trains a neural representation model by leveraging tremendous amounts of unlabeled smart contracts. SRCL starts by converting the abstract syntax tree (AST) of each smart contract into a pair of type and value sequences using pre-order traversal. The type sequence involves structural information while the value sequence retains semantic information. Then, three operations are performed upon the pairs of sequences in order to increase the scale and diversity of training samples.

Based on the augmented code variants, SRCL trains a Transformer encoder which encodes smart contract code into vectors through three components: a local discriminator which assists the encoder to capture lexical and syntactical features, a global discriminator which enables the encoder to learn global semantics of code, and a decoder which aims to reconstruct the value sequence of the input code.

To evaluate the effectiveness of SRCL in learning smart contract representations, we collect 75,006 smart contracts from Etherscan[2].

We extensively evaluate SRCL on three downstream tasks, namely, bug detection, clone detection, and code clustering, and compare the performance against three recently proposed approaches, including SmartEmbed [14], code2vec [2], and code2seq [1]. Experimental results show that SRCL outperforms baseline models significantly. It improves F1-scores of the three tasks by 6.96%, 4%, and 8.81%, respectively.

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

- We propose a novel approach for learning smart contract representations, which leverages three self-supervised learning tasks for capturing both global and local semantic information of source code.
- We propose three operations for generating code variants, namely, *type replacement*, *value replacement*, and *pair insertion* to increase the scale and diversity of training samples.
- We build a dataset for clone detection and code clustering tasks of smart contracts.

The dataset and source code of SRCL are publicly available at: https:// github.com/SCRepslearner/SmartLearner.

2 BACKGROUND

2.1 Smart Contract

Smart contracts are general-purpose computer programs that run on Ethereum. Smart contracts have been widely applied to many fields such as currency trading platforms, crowdfunding campaigns, and role-playing games. A smart contract is a series of instructions or operations which will be triggered when certain conditions are satisfied. Smart contracts achieve a great success because they eliminate the need of trusted third parties in multiparty interactions so that parties can engage in secure peer-to-peer transactions without having to place trust in external parties.

Figure 1 shows a basic structure of smart contract. Generally, a smart contract is consists of four elements: a unique address identifies the contract, a set of executable functions, state variables, and values [4]. It takes as input transactions with function parameters, executes the corresponding code and triggers the output events. Upon the execution of a function, the state variables in the contract change according to the logic implemented in the function.

In Ethereum, smart contracts are primarily written in Solidity, which is a high-level contract-oriented programming language.

[2]http://etherscan.io/

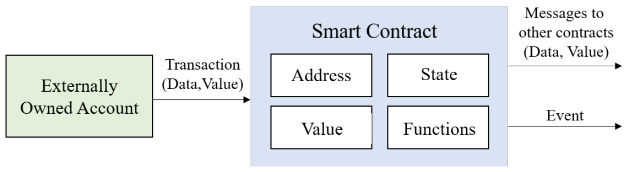


Figure 1: The basic structure of smart contracts [47].

Source code in Solidity can be compiled to bytecode run on the Ethereum Virtual Machine (EVM). The syntax of Solidity resembles that of general object-oriented languages such as Java and C++. A source file of Solidity may contain three key structures, namely, *libraries*, *interfaces*, and *subcontracts*. A *library* is a piece of code that provides a set of common functionalities that are mindful of corner-cases or that optimize processing time. Library is stateless and doesn't contain state variables. Analogous to the interface in Java, an *interface* in Solidity is an abstract contract which does not have any implemented functions. A *subcontract* is a contract that implements a certain concepts. It contains state variables and functions for accessing and modifying the state variables. In solidity, a *subcontract* often inherits and realizes an *interface*.

2.2 Data Augmentation

Data augmentation (DA) aims to increase both the amount and diversity of a dataset without explicitly collecting more data[11]. DA has become a practical technique in computer vision and natural language processing (NLP) tasks that have low resources and a paucity of annotated data [11, 36]. DA is often regarded as a regularizer to reduce the risk of overfitting when training deep learning models. Due to its effectiveness, DA has been increasingly leveraged as a key technique of self-supervised learning.

DA techniques can be roughly classified into three categories, namely, rule-based approaches, model-driven approaches, and example interpolation approaches [11].

The rule-based approaches directly apply slight modifications that follow some heuristic rules on the copies of the original samples. For example, the EDA model [52] improves text classification by token-level perturbation, including *synonym replacement*, *random insertion*, *random swap*, and *random detection*. Sahin *et al.* [39] augmented the training sets of low-source languages by cropping and rotating the dependency tree of sentences.

Model-based approaches augment samples by using a well-trained neural network. The typical model-based DA technique is back-translation [44] which translates original sentences into intermediate languages and then back translates them to paraphrases. Cai *et al.* [7] proposed an end-to-end learnable data manipulation model for augmenting effective training samples and reducing the weights of inefficient samples. This model has been shown to be effective in boosting dialogue systems.

The idea of example interpolation comes from Mixup [56]. Given two pairs of real samples and their one-hot labels: (x_i, y_i) and (x_j, y_j), Mixup constructs a new virtual sample $(\tilde{x}, \tilde{y})$ by incorporating them together with a parameter λ:

$$\begin{aligned} \tilde{x} &= \lambda x_i + (1-\lambda)x_j \\ \tilde{y} &= \lambda y_i + (1-\lambda)y_j \end{aligned} \tag{1}$$

Inspired by Mixup, Guo *et al.* [16] proposed wordMixup and sentMixup for sentence classification that interpolate samples in the word and sentence embedding spaces, respectively.

2.3 Self-Supervised Learning

As the most common technique for training neural networks, supervised learning has its own bottleneck since it relies heavily on manual labeled data. As an alternative, self-supervised learning (SSL) shows its potential in leveraging the tremendous amounts of unlabeled data. SSL has drawn much attention for its broad applicability on various machine learning tasks such as image recognition [17, 26], pre-trained language models [9, 29, 54], and graph representation learning [10, 50]. The goal of SSL is to predict partial attributes of an object from remaining parts. Given an object x, SSL usually trains an encoder which encodes x into an explicit vector z by predicting an automatically generated label for a predefined self-supervised task. In such a way, the explicit vector z can be leveraged by downstream tasks, since it contains universal features from x that benefit to different tasks.

It is critical to choose proper self-supervised tasks in SSL, since they enable the encoder to learn semantics of objects. In natural language processing, there are a variety of self-supervised tasks. As a typical SSL-based technique, Word2vec [35] learn word embeddings by predicting a word from its contexts within a certain size of window. BERT [9] utilizes the masked language model as a self-supervised task by predicting the masked words in a sentence. WKLM [53] adopts the replaced entity detection task and trains a knowledge-enhanced pre-trained language model for entity-related question answering. XLNet [54] introduces a permutation language model, which combines the strengths of both auto-regressive language model and auto-encoder language model. XLNet learns bidirectional contexts by maximizing the expected likelihood over all permutations of the factorization order. InfoWord [28] is proposed to learn language representations by maximizing the mutual information between a representation of a sentence and an n-gram within it.

3 APPROACH

3.1 Overview

Figure 2 illustrates the overall framework of SRCL. The pipeline involves three phases, namely, constructing structural sequences, generating code variants, and representation learning. Initially, we extract structural sequences from smart contract ASTs to facilitate deep representation learning of structures. Then, we augment the processed contracts by generating code variants using three operations, namely, type replacement (TR), value replacement (VR), and pair insertion (PI). Subsequently, we design a self-supervised representation model which takes as input the augmented structural sequences and outputs the code vectors. Finally, We leverage the generated code vectors to assist three crucial downstream tasks for smart contracts.

3.2 Constructing Structural Sequences

The main challenge of learning code representations is how to incorporate structural information in source code [19, 20, 57]. Structures are significant features for modeling source code [19] as they specify

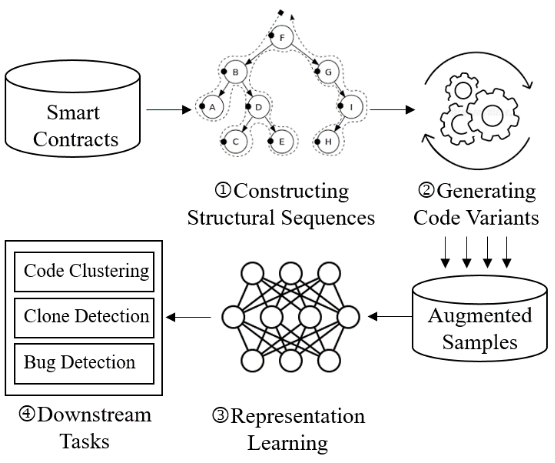


Figure 2: The framework of SRCL.

how different statements interact with each other to accomplishing certain functionality. Simply treating source code as plain texts will lose the additional semantics to the program functionality besides the lexical terms, resulting in inferior performance [20].

In order to learn both semantic and syntactic information in smart contracts, our approach starts from extracting structures from smart contracts. We convert each smart contract into two sequences, namely, AST type sequence and AST value sequence by traversing its modified AST. Such sequential representations of structural data can be easily processed by modern sequence learning models such as the Transformer [25].

Figure 3 shows the detailed process. Given a smart contract, we initially build its AST. Each node in the AST contains two elements, namely, *node type* and *node value*. The *node value* is the concrete token occurring in the source code while the node type is its abstract type. For example, the root node in Figure 3 has a *node type* “ContractDefinition" with its contract name “shapeCalculator” as the corresponding *node value*.

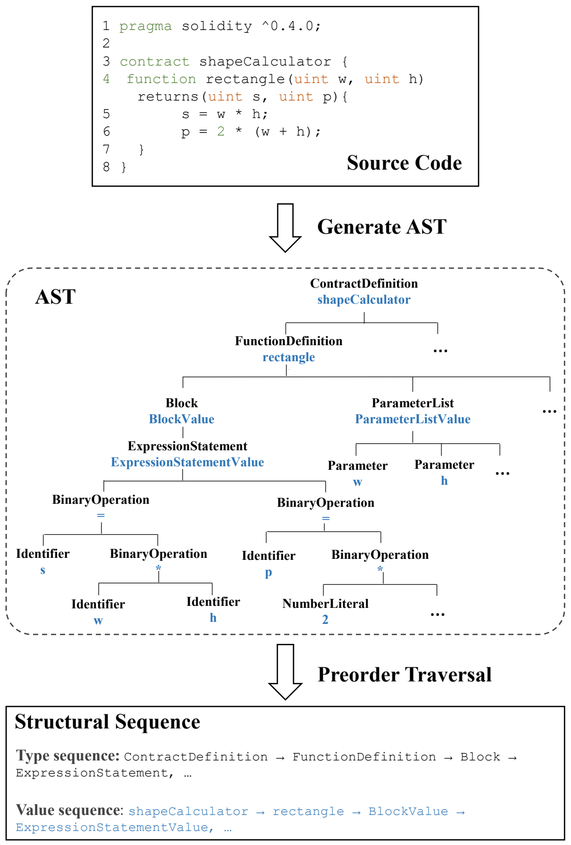
For non-terminal (NT) nodes that have no information of node value, default values will be padded according to their types. For example, if an NT node has a *type “FunctionCall”* and has no corresponding *value*, we will assign it a default *value “FunctionCallValue”*.

Next, we generate two structural sequences (i.e., the type sequence and the value sequence) through a pre-order traversal on the AST. These sequences can be used as structural information of smart contract source code for further representation learning.

3.3 Generating Code Variants

Another challenge of learning code representations lies in the homogeneity of smart contract source code. In order to capture the deep semantic features in smart contracts, we need large amounts of variants for training a deep representation model.

Inspired by previous works on data augmentation [51, 52], we propose three code variants generation operations. More specifically, given all structural sequences, we maintain a key-value map M for all the type tokens and the value tokens, where the



```
pragma solidity ^0.4.0;

contract shapeCalculator {
 function rectangle(uint w, uint h)
  returns(uint s, uint p){
       s = w * h;
       p = 2 * (w + h);
  }
}
```

Structural Sequence

Type sequence: `ContractDefinition → FunctionDefinition → Block → ExpressionStatement, …`

Value sequence: `shapeCalculator → rectangle → BlockValue → ExpressionStatementValue, …`

Figure 3: An illustration of constructing structural sequences for smart contracts.

key is a *type* token and the values are its corresponding *value* tokens from the ASTs. Given a sample $S =< T, V >$ in our training set, where $T = (t_1, t_2, ..., t_n)$ represents the type sequence, and $V = (v_1, v_2, ..., v_n)$ represents the value sequence, we perform the following three operations, each repeated K times:

1) *Type Replacement (TR):* we randomly choose a token from the type sequence and replace it with another random *type* token from M.
2) *Value Replacement (VR):* we randomly choose a token v_i from the value sequence and replace it with another *value* token $v_j (i \neq j)$ from M, where v_i and v_j associated with the same type token.
3) *Pair Insertion (PI):* we find a random pair $< t_i, v_j >$ from M and insert it into a random position of the sample.

We hypothesize that long sequences can absorb more noise while maintaining their original semantic information. Therefore, the number of value tokens changed, K, is calculated by the formula $K = \alpha l$, where l is the length of the sequence, and α indicates the percent of the tokens or pairs in a sequence are changed. Compared with the original sample, *TR* and *PI* change the syntax and structural information, while *VR* maintains the original syntax and structures and changes the semantics of original sample.

All the generated variants, together with the original training samples, are taken as input to the representation learning model.

3.4 Representation Learning

We design a neural network model for learning representations of smart contracts from the structural and sequential input as well as the sufficient variants of code. Figure 4 shows its architecture overview. The model mainly consists of four components, namely, encoder, local discriminator, global discriminator, and decoder.

The learning process involves three steps. First, the encoder takes as input the structural sequences of a smart contract and generates the local representation using the Transformer [49] encoder. The encoded local representation is aggregated into a global representation using a convolutional neural network (CNN) [13]. Second, local and global discriminators are applied to the local and global representation respectively and discriminate whether each token or the whole semantic has been changed (*i.e.*, *token replaced prediction task* and *real-fake sample prediction task*), so as to learn the local and global features of a smart contract. Third, the decoder takes as input the global representation and learns to reconstruct the original value sequence (*i.e.*, *value sequence recovery task*).

Transformer Encoder. Let $S =< T, V >$ denotes a smart contract in the form of structural sequences, where $T = t_1, ..., t_n$ stands for the type sequence and $V = v_1, ..., v_n$ stands for the value sequence. The Transformer encoder takes S as input, and embeds each of the token in T and V into a d-dimensional vector.

$$\begin{aligned} \boldsymbol{e}(T) &= [\boldsymbol{e}(t_1), ..., \boldsymbol{e}(t_n)] \\ \boldsymbol{e}(V) &= [\boldsymbol{e}(v_1), ..., \boldsymbol{e}(v_n)] \end{aligned} \tag{2}$$

The type and value embeddings are summed up into one sequence:

$$\boldsymbol{x} = [\boldsymbol{x}_1, ..., \boldsymbol{x}_n], \tag{3}$$

where $\boldsymbol{x}_i = \boldsymbol{e}(t_i) + \boldsymbol{e}(v_i)$.

Then the Transformer encoder generates the local representations from $\boldsymbol{x}$:

$$\boldsymbol{R}_{\text{local}} = [\boldsymbol{H}_1, ..., \boldsymbol{H}_n] = \text{Enc}(x_1, ..., x_n), \tag{4}$$

where each $\boldsymbol{H}_i \in R^d$ stands for the hidden state of the encoder for the i^{th} input token.

Subsequently, a CNN is applied to summarize the local representations into global representation, as shown in Figure 5. Each filter in the CNN generates a feature map from the local representation:

$$\begin{aligned} \boldsymbol{c}_i &= \text{ReLU}(\boldsymbol{W} \cdot \boldsymbol{H}_{i:i+k-1}), \\ \boldsymbol{f} &= \text{maxpooling}([\boldsymbol{c}_1, \boldsymbol{c}_2, ..., \boldsymbol{c}_n]), \end{aligned} \tag{5}$$

where *ReLU* is an activation function, $\boldsymbol{W}$ is a trainable parameter matrix, c_i stands for a feature over a window of k vectors from $\boldsymbol{R}_{\text{local}}$, and $\boldsymbol{f}$ stands for the most important feature in the feature map.

Finally, all feature maps are concatenated and passed into a linear layer, yielding the global representation.

$$\boldsymbol{R}_{\text{global}} = \boldsymbol{W} \cdot [\boldsymbol{f}_1; ...; \boldsymbol{f}_n] \tag{6}$$

Global Discriminator. Given the global representation of a smart contract, a global discriminator is designed to perform the *real-fake sample prediction task*, which predicts whether the contract is

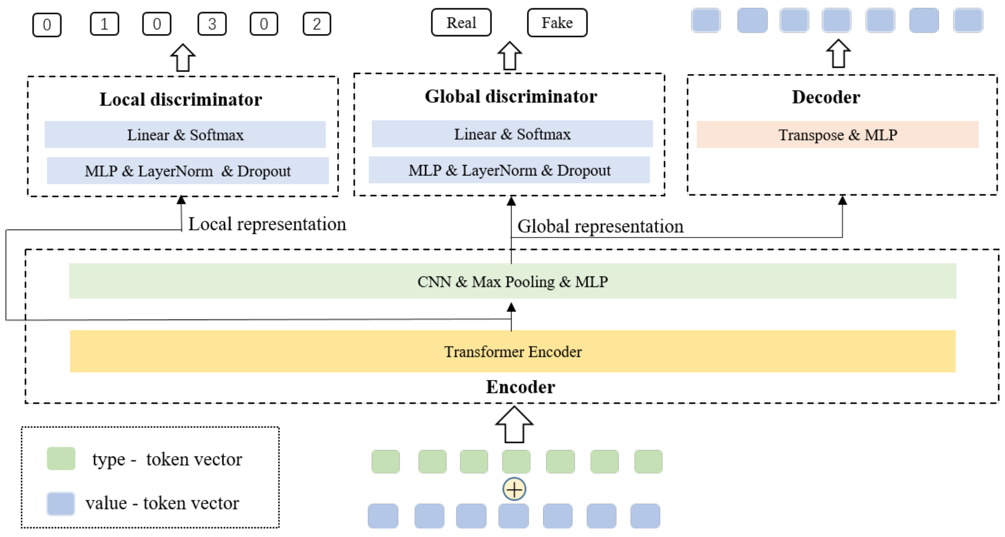


Figure 4: The architecture of our representation learning model.

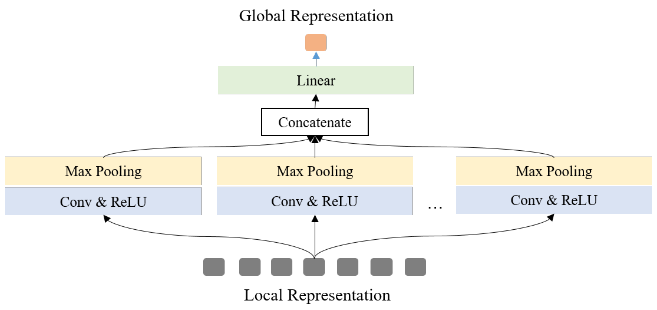


Figure 5: The generation of global representation.

syntactically changed. We formulate this as a classification problem where either a positive sample (the original sample or sample created by *VR*) or a negative sample (sample created by *TR* and *PI*) of contract is provided to the discriminator. The global representation $\boldsymbol{R}_{\text{global}}$ is passed to an MLP classifier and the probability distribution over labels is returned.

$$\tilde{y}^g = \text{softmax}(\text{ReLU}(\text{LayerNorm}((\boldsymbol{R}_{\text{global}} \cdot \boldsymbol{W}_1)...\boldsymbol{W}_n)), \quad (7)$$

where $\boldsymbol{W}_i \in \mathbb{R}^{\text{dxd}}$ represents the weight matrix for the i^{th} fully-connected layer, *LayerNorm* stands for layer normalization, and *ReLU* is the activation function.

The global discriminator is trained to minimize the following cross-entropy loss of *real-fake sample prediction task*:

$$\mathcal{L}_g = -(\log y^g(\tilde{y}^g) + (1 - y^g) \cdot \log(1 - \tilde{y}^g)), \quad (8)$$

where y^g and $\tilde{y}^g$ are the true label and probability.

Local Discriminator. Apart from the global discriminator which judges the changing state of the whole contract, we also design a local discriminator to accomplish the *token replaced prediction task*, which predicts for each token whether it is changed or not and what is the type of the change (i.e., *TR*, *VR* or *PI*).

The local discriminator is also implemented with an MLP classifier. It takes as input the local representation at each position of the contract and classifies the type of change for the position:

$$\tilde{y}^l = \text{softmax}(\boldsymbol{R}_{\text{local}'} \cdot \boldsymbol{W}_o), \quad (9)$$

where $\tilde{y}^l$ stands for the probability distribution over labels, and $\boldsymbol{W}_o$ stands for the trainable parameter matrix.

Similarly, the local discriminator aims to minimize the following cross-entropy loss of *token replaced prediction task*:

$$\mathcal{L}_l = -\frac{1}{N} \sum_{i=1}^{N} \sum_{j=1}^{C_l} y^l_{i,j} \log(\tilde{y}^l_{i,j}), \quad (10)$$

where $y^l_{i,j}$ and $\tilde{y}^l_{ij}$ denote the true label and the probability of the i^{th} token belonging to the j^{th} category, respectively; C_l represents the number of categories; and N represents the length of the input sequence.

Decoder. The decoder aims to complete the *value sequence recovery task*, which reconstructs the value sequence from the global representation. Similar to the global discriminator, we first pass the global representation to a fully-connected neural network, and obtain the output $\boldsymbol{R}'_{global} \in \mathcal{R}^{n\times m}$. Subsequently, we apply a *transpose* operation on $\boldsymbol{R}_{\text{hidden}}$: $\boldsymbol{R}'_{\text{global}} \in \mathcal{R}^{n\times m} \rightarrow \boldsymbol{R}'_{\text{global}} \in \mathcal{R}^{m\times n}$ and put it into a fully-connected neural network to obtain the probability distribution over labels:

$$\begin{aligned} \boldsymbol{R}'_{\text{global}} &= \text{ReLU}(\text{LayerNorm}((\boldsymbol{R}'_{\text{global}} \cdot \boldsymbol{W}_1 + b_1)...\boldsymbol{W}_n), \\ \tilde{y}^d &= \text{softmax}(\boldsymbol{R}_{global'} \cdot \boldsymbol{W}_o). \end{aligned} \quad (11)$$

The decoder is trained to minimize the following loss function of *value sequence recovery task*:

$$\mathcal{L}_d = -\frac{1}{N} \sum_{i=1}^{N} \sum_{j=1}^{C_d} y^d_{i,j} \log(\tilde{y}^d_{i,j}), \quad (12)$$

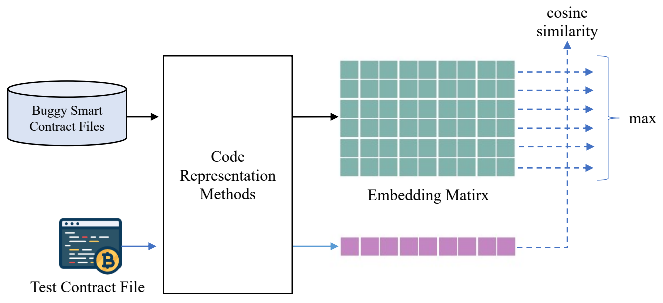


Figure 6: The process of bug detection.

where $y^d_{i,j}$ and $\tilde{y}^d_{i,j}$ denote the true label and the probability of the i^{th} token in the j^{th} category, respectively; and C_d represents the number of categories.

Model Training. We train our model by minimizing the total loss, which is defined as the weighted sum of the losses of three self-supervised learning tasks [31]:

$$\mathcal{L} = \frac{1}{2\sigma_g^2}\mathcal{L}_g+\frac{1}{2\sigma_l^2}\mathcal{L}_l+\frac{1}{2\sigma_d^2}\mathcal{L}_d+\log(1+\sigma_g^2)+\log(1+\sigma_l^2)+\log(1+\sigma_d^2), \tag{13}$$

where $\mathcal{L}_g$, $\mathcal{L}_l$, and $\mathcal{L}_d$ represent the loss functions for the three tasks, respectively; σ^g, σ^l, and σ^d are their weighting factors, which are automatically learned in the training process.

3.5 Downstream Tasks

We test our SRCL on three downstream tasks, namely, bug detection, code clone detection, and code clustering. They are all well known tasks for evaluating code representations and are also critical tasks for smart contracts.

Bug Detection. Programmers tend to copy and paste existing smart contract code rather writing it from scratch, which may introduce clone-related bugs into programs. Therefore, this task aims to detect whether a given code snippet is "similar" to any known bug [14]. Figure 6 shows the process of bug detection. First, we construct a bug embedding matrix of known buggy smart contract files using representation models. Then, given a smart contract file to be detected, we obtain its code vector and calculate its similarity with each vector in the bug embedding matrix. Since each file contains several subcontracts, we measure the similarity between two files using the "group similarity" between their subcontracts, namely,

$$sim(A,B) = \frac{\sum_{i=1}^{M} max(\{sim(A_i,B_j) \mid 1 \le j \le N\})}{M}, \tag{14}$$

where M and N are the numbers of subcontracts in file A and file B, and A_i and B_j stand for two individual subcontracts in the two files, respectively; sim denotes the cosine similarity function. If the similarity between the given contract and a known buggy contract exceeds a predefined threshold, then the contract is reported as a potential bug.

Code Clone Detection. The code clone detection task aims to detect whether two smart contracts are semantically identical, *i.e.*, implement the same functionality [57]. Given a pair of smart contracts, we predict them as a cloned pair if the similarity between their code vectors is greater than a predefined threshold.

Code Clustering. The code clustering task aims to cluster all smart contracts (without class labels) according to the pairwise similarities between their code vectors. We cluster all contracts into K clusters using the k-Means [23] algorithm. Next, we measure the correctness of the output clusters by calculating the ARI score [41] based on the real class labels of these smart contracts.

Table 1: Statistics of the dataset for learning code representations.

original	# contracts	55,006
	# distinct subcontracts	290,390
	# statements	14,673,335
	avg lines per distinct subcontract	59.47
augmented	# samples	1,161,560
	avg sequence length	185.73

Table 2: Statistics of datasets for code clone detection and code clustering tasks.

Clone detection	# clone pairs	3457
	# avg code lines per subcontract	116
Code clustering	# clusters	119
	# avg subcontracts per cluster	258.94
	# avg code lines per subcontract	60.32

4 EXPERIMENTAL SETUP

We conduct several experiments with the aims of investigating the following research questions (RQs):

- RQ1: How effectively does our SRCL perform under three downstream tasks, compared with the state-of-the-art methods?
- RQ2: How do internal components or techniques contribute to the effectiveness of SRCL?

4.1 Datasets

Dataset for learning code representations. We collect verified smart contracts from the Ethereum's block explorer and analytic platform - Etherscan[3] using web scrapers built by ourselves. Table 1 shows the statistics of the training dataset. We have gathered 75,006 verified smart contracts written in Solidity language. 55,006 of them are used for training while the remaining 20,000 are used as the datasets for the code clone detection and code clustering tasks.

Since a smart contract usually consists of a number of subcontracts, we extract subcontracts from each Solidity file. There are totally 290,390 distinct subcontracts in the dataset. Each subcontract involves an average of 59.47 lines of code. After generating code variants, the training data is augmented to 1,161,560 samples, and each sample contains an average of 185.73 type tokens.

Datasets for clone detection and code clustering. We then construct datasets for clone detection and code clustering from the

[3]https://etherscan.io/

Table 3: Statistics of known buggy smart contracts for bug detection task.

category	A				B			
# contracts	227				319			
type	A2	A6	A10	A16	B1	B4	B5	B7
# contracts	53	32	67	75	56	30	183	50

remaining 20k smart contracts. We extract subcontracts from each Solidity file and classify them into different categories by their contract names. Small categories (i.e., $\leq$ 100 subcontracts) are used for clone detection while large categories (i.e., $>$ 100 subcontracts) for code clustering. To construct the clone detection dataset, we randomly select a pair of contracts from each of small categories as a true clone pair, and randomly sample an equal number of false clone pairs from different categories. Both datasets are then manually checked to reduce noise. Table 2 shows the statistics of the final datasets for code clone detection and code clustering. Overall, we collect 3,457 true clone pairs for the clone detection task and 119 large categories are selected for code clustering.

Dataset for bug detection. For evaluating the bug detection task, we use smart contracts from the Awesome Buggy ERC20 Tokens[4]. The dataset is a collection of vulnerabilities in ERC20 smart contracts collected from public resources, which have been manually checked by 9 contributors. These bugs are divided into 29 types, and further grouped into three categories: A) bugs in implementation, B) incompatibilities caused by different compiler versions and external calls, and C) excessive authorities. Since the quantities of some types of buggy contracts are small, we filter out types that contain no more than 30 contracts from the dataset. We also sample the same amount of validated smart contracts from OpenZeppelin[5], a library for secure smart contract development. They are regarded as "bug-free" smart contracts and can help identify the rate of false positive and false negative samples. Each bug that is discovered among them is automatically considered as a false positive. Table 3 describes the statistics of known buggy smart contracts for bug detection task. In total, 227 smart contracts of category A and 319 smart contracts of category B are used in our evaluation. For each type of the buggy smart contract files, we divide them equally into two subsets: half of the buggy files are used to construct a bug embedding matrix, while the other half are taken as the test set.

4.2 Baselines

We compare our approach with three state-of-the-art methods on code representations learning: SmartEmbed [14], code2vec [2] and code2seq [1]. Smartembed is the most advanced smart contract representation method. Code2vec and code2seq are state-of-the-art methods for general programming languages. We do not compare our method with information retrieval based approaches (*e.g.* TF-IDF and N-Gram) because our baseline models have demonstrated great improvement over information retrieval based methods.

1) SmartEmbed: the latest approach that is specifically designed for learning smart contract representations. SmartEmbed parses smart contract code blocks into word streams and converts them into numerical vectors by word embedding techniques. Then, it identifies smart contracts that are correlated to known bugs by their vector similarities.

2) code2vec: an approach to learn representations of general programming languages. code2vec trains a path encoder on bag-of-paths that are extracted from ASTs. The path encoder encodes the selected paths into a single, fixed-length *code vector*. Then, the generated *code vector* is taken as input to a classifier which predicts the method name. In this way, the model learns useful *code vectors* that capture semantic similarities, combinations, and analogies.

3) code2seq: another general approach for learning code representations. Similar to code2vec, code2seq represents a code snippet as a set of compositional paths in its AST. Next, it uses the attention mechanism to select the relevant paths. The selected paths are encoded into a vector and further decoded to a natural language summary of the code snippet.

Since code2vec and code2seq are originally implemented for Java, we adapt it to Solidity in our work. More specifically, we change their inputs to the same format as our smart contracts. We follow their data processing steps to prepare the inputs. The number of sampled paths from the AST of each individual contract is 100 for both code2vec and code2seq, which is a good sweet point between capturing enough contract information while keeping training feasible in the GPU's memory. The dimensions for word embedding in code2vec, code2seq and SmartEmbed are set to 128, 128 and 150 respectively, following their original setups.

It is worth noting that we do not compare our approach with pre-trained models such as CodeBERT[12], because they need large scale parallel corpora of programming and natural languages for pre-training, which is inapplicable in the field of smart contracts.

4.3 Implementation Details

We implement our approach based on PyTorch 1.4 and Python 3.6. To convert smart contracts into ASTs, we use a Solidity parser[6] for Python which is built on top of a robust ANTLR4 grammar.

The vocabulary sizes for the type and value tokens are 94 and 20,529, respectively. The embedding dimensions for type tokens and value tokens are set as 256. The encoder of SRCL contains six Transformer layers with eight attention heads. The CNN layer consists of four convolution kernels with different sizes (i.e., 3, 5, 7, 9). The hyperparameters for measuring the importance of the three learning tasks are $\sigma^g = \sigma^l = \sigma^d = 0.5$. All models are optimized using the Adam algorithm with a learning rate of 0.001 and a dropout ratio of 0.5. The similarity threshold for bug detection and clone detection is empirically set to 0.95. SRCL is trained for 5 epochs with a batch size of 64. All models are run on a server with a GeForce GTX 1080 Ti GPU, 64 GB memory, and Ubuntu 18.04.

4.4 Metrics

We measure the performance of the bug detection and clone detection tasks using the well-known precision, recall, and F1-score. For the code clustering task, we use the Adjusted Rand Index (ARI)[41], which measures the degree of agreement between two data partitions. Let U be the ground truth class assignment, and V be the number of clusters yielded by a clustering algorithm, the ARI in

[4]https://github.com/sec-bit/awesome-buggy-erc20-tokens
[5]https://github.com/OpenZeppelin/
[6]https://github.com/ConsenSys/python-solidity-parser

Table 4: Performance of various approaches in the bug detection task.

Approach	Precision	Recall	F1-score
code2vec [2]	0.4144	0.3970	0.3998
SmartEmbed [14]	0.5372	0.4934	0.5017
code2seq [1]	0.5180	0.5573	0.5323
SRCL	**0.6266**	**0.6196**	**0.6019**

our evaluation is defined as follows:

$$ARI = \frac{RI - E(RI)}{\max(RI) - E(RI)}$$
$$RI = \frac{a+b}{a+b+c+d} \tag{15}$$

where a is the number of element pairs that are in the same clusters in U and the same clusters in V, b is the number of of pairs of elements that are in the same clusters in U but different clusters in V, c denotes the number of pairs of elements that are in different clusters in U but same clusters in V, d denotes the number of pairs of elements that are in different clusters in U and different clusters in V. The range of ARI is -1 between 1, and a higher ARI would indicate a better clustering result.

5 RESULTS

5.1 Effectiveness in Bug Detection

Table 4 shows the performance of SRCL and baselines in the bug detection task. We can observe that SRCL achieves the best performance compared to baseline models. For example, the F1-score obtained by SRCL is 0.6019, which is significantly greater than that of code2vec (0.3998), SmartEmbed (0.5017), and code2seq (0.5323).

code2seq outperforms code2vec, probably because code2vec only represents AST paths, while code2seq represents both internal and terminal nodes in ASTs. SmartEmbed embeds terminal nodes using FastText [5]. However, the embeddings are not further integrated into vectors of code fragments, which restricts the ability of SmartEmbed in representing bug relevant tokens. Compared to these approaches, SRCL incorporates both type and value information in ASTs and explicitly integrates them through a Transformer encoder for capturing both lexical and syntactical knowledge of smart contracts.

Furthermore, code2vec and code2seq train their encoders using the contract name generation task. However, unlike general programming languages, the name (*e.g.* Token, ERC20 and StandardToken) of a smart contract usually does not present its semantics.

By contrast, the self-supervised tasks in SRCL take into account both local and global semantics of smart contracts and enable a better learning of smart contract representations.

5.2 Effectiveness in Clone Detection

Table 5 shows the performance of various approaches in the code clone detection task. Overall, SRCL obtains the best results compared to baseline models. For example, the recall and F1-score obtained by SRCL are 0.5908 and 0.6185, which are significant greater than those of SmartEmbed (0.4241 and 0.4617), code2vec (0.5269 and 0.5105) and code2seq (0.5562 and 0.5789).

Table 5: Performance of various approaches in the code clone detection task.

Approach	Precision	Recall	F1-score
SmartEmbed [14]	0.6698	0.4241	0.4617
code2vec [2]	0.7395	0.5269	0.5105
code2seq [1]	**0.7990**	0.5562	0.5789
SRCL	0.7955	**0.5908**	**0.6185**

Table 6: Performance of various approaches in the code clustering task.

Approach	ARI
SmartEmbed [14]	0.6260
code2vec [2]	0.5365
code2seq [1]	0.6631
SRCL	**0.7512**

One reason for this could be that code2vec and code2seq represent two AST paths that have minor difference as distinct vectors. This may result in incorrect classifications in syntactic clones which are merely different in identifiers and comments.

SmartEmbed achieves the worst performance. We conjecture that it does not explicitly represent global semantics, thus restricting the detection of semantic clones. Comparatively, SRCL measures the semantic similarities between smart contracts by learning to represent both local syntax and global semantics.

5.3 Effectiveness in Code Clustering

Table 6 presents the results of various approach in the code clustering task. As can be seen, SRCL achieves the best performance in terms of ARI (0.7512), followed by code2seq (0.6631) and SmartEmbed (0.6260). This indicates that SRCL can successfully cluster smart contracts with efficacy.

To further analyze their performance, we visualize the code vectors produced by program representation learning methods. More specifically, we randomly select vectors produced by the four methods for 10 classes of contracts on the cluster dataset. Then, we map each vector into a two-dimensional space using T-SNE [48].

Figure 7 plots the visualization of these two-dimensional vectors. We can observe that vectors generated by SRCL has clear boundaries than those generated by other methods. This facilitates the K-Means algorithm to cluster contracts. We further observe that vectors of some classes (*e.g.* red) are close to each other, nearly clustered as a point. This confirms our initial finding that code reuse is a common phenomenon in smart contracts.

5.4 Ablation Study

To get a better insight into SRCL, we perform an in-depth ablation study on the code clone dataset. The main goal is to validate the effectiveness of the critical components or techniques in our architecture including the global discriminator, the local discriminator, the decoder, and the code variant generation module.

Table 7 shows the results of all variants of our model. As seen, all the studied components contribute to the effectiveness of SRCL

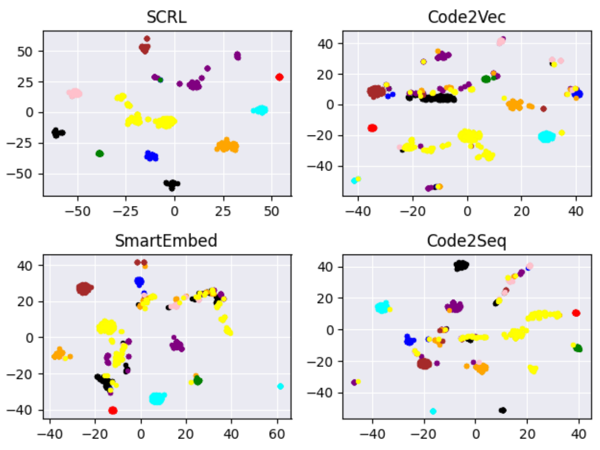


Figure 7: Visualization of the code vectors produced by different program representation learning methods.

Table 7: Results of ablation studies.

Model	F1-score	△
SRCL (original model)	**0.6681**	-
w/o global discriminator	0.5883	-0.0798
w/o local discriminator	0.6427	-0.0254
w/o decoder	0.6243	-0.0483
w/o type replacement	0.6487	-0.0194
w/o value replacement	0.6415	-0.0266
w/o pair Insertion	0.6575	-0.0106

in learning smart contract representations. In particular, the global discriminator contributes the most in terms of F1-score. This is because the global discriminator is directly correlated to the global representation of smart contracts. Similarly, the reconstruction objective of the decoder is also effective as it forces the encoder to learn key features of smart contracts. The local discriminator also helps SRCL in learning local syntactic representations.

The three operations for creating code variants are also effective in learning code representations. The *value replacement* operation does not significantly change the syntax and structure of the original code, while the *type replacement* and *pair insertion* operations considerably modify the original code, reducing the risk of overfitting. To further understand the importance of generating code variants, we conduct an experiment on various fractions of the training data. Figure 8 shows the results. We can see that the generation of code variants is more effective on smaller training sets. This is because small datasets are more likely to overfit training data than larger datasets. As the scale of dataset increases, the improvement becomes less significant.

5.5 Summary

Across all the experiments, SRCL significantly outperforms existing methods in learning smart contract representations. For example, SRCL outperforms code2seq by 6.96% and 4% in terms of F1-score

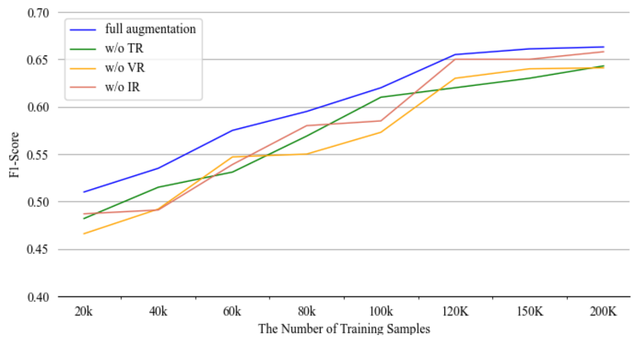


Figure 8: Impacts of different code variant generation techniques in the bug detection task.

in the bug detection and clone detection tasks, respectively. SRCL also outperforms code2seq by 8.81% in terms of ARI in the code clustering task.

All three components contribute to the effectiveness of SRCL. Among them, the global discriminator has the most significant effect. Generating code variants also enhances the performance of SRCL, especially when the scale of training data is small.

Overall, the experimental results suggest that SRCL has remarkable effectiveness in learning smart contract representations.

6 THREATS TO VALIDITY

We have identified two main threats to the validity.

Data quality threat. The test sets we have used for code clone detection and code clustering are built by matching contract names. Although we have manually checked the similarities between clone pairs, there can still be noise. We leave further refinement of the test set for future work.

Genericity threat. As our work focuses on learning smart contract representations, we only test SRCL on the Solidity language. Although Solidity is one of the most popular languages for smart contracts, other programming languages could have different results. In the future, we will conduct more extensive evaluation of our approach on other smart contract languages (*e.g.* Serpent and LLL) and other programming languages (*e.g.* Go and VHDL).

7 RELATED WORK

7.1 Learning Program Representations

Learning program representations has been a fundamental problem in software engineering [1, 2, 6, 57]. Existing approaches for source code representation fall into two categories, namely, information retrieval (IR) based approaches and deep learning (DL) based approaches.

The IR based approaches treat source code as plain texts and employs various information retrieval techniques for specific software engineering tasks. For example, Deckard [22] learns syntax-structured information from source code for clone detection. SourcererCC [40] is a token based detector which improves code clone detection for very large code bases by using an optimized inverted index technique. Lukins *et al.* [32] showed that the performance of LDA-based fault localization model is not affected by the size of subject software system. Kim *et al.* [24] employed naive Bayes to match bug reports with source files for bug localization.

Since DL techniques have achieved a great success in NLP, DL based code representation approaches have attracted much attention. Nix *et al.* [37] applied a multi-layer CNN classifier to detect malicious software programs in mobile apps. Gu *et al.* [15] proposed DeepCS which jointly represents natural language and source code (API invocation sequence, token sequence, and method name) using recurrent neural networks and multi-layer perceptions. Li *et al.* [30] proposed CQIL to learn code-query interactions, which uses a CNN to compute semantic correlations between queries and code snippets. Zhou *et al.* [58] presented a context-aware code-to-code recommendation tool named Lancer, with the support of a Library-Sensitive Language Model (LSLM) and the BERT model. ASTNN [57] decomposes a large AST of a code snippet into small statement trees, recursively encodes multi-way statement trees to obtain their vectors, and learns the code representation by following the naturalness of statements. In these DL based approaches, the use of deep neural networks significantly improves the understanding of code semantics, thereby showing better effectiveness. However, they are supervised and rely on the availability of labeled data for training. Since Solidity is a domain-specific language and suffers from the shortage of labeled training data, these models are not applicable to smart contracts. Besides, they are usually trained for the specific tasks, lacking generality to support multiple tasks with one single model.

Most recently, pre-trained programming language models have achieved a great success. CodeBERT [12] pre-trains a Transformer-based neural architecture for programming languages and natural languages to learn general-purpose representations. ContraCode [21] pre-trains a neural network to identify functionally similar variants of a program among many non-equivalent distractors. InferCode [6] pre-trains a tree-based CNN encoder for source code representation by predicting subtrees from the context of ASTs. All these pre-trained models have shown significant effectiveness in learning code vectors from major programming languages (*i.e.*, Java, C). However, it is difficult for them to achieve the equally effect in smart contracts, since smart contracts are highly homogeneous and redundant. Such data redundancy cause models to be poorly fitted especially on insufficient training samples.

Different from these works, our SRCL takes into consideration the unique characteristics of smart contracts, and captures both local (*e.g.*, structural, lexical) and global (*e.g.*, semantics) features from unlabeled smart contracts. SRCL designs three data augmentation operations for increasing the diversity of smart contracts in training set, thus having the ability of learning representations in the case of small training samples.

7.2 Deep Learning in Smart Contracts

In recent years, there is an emerging trend in applying deep learning to smart contracts [34, 55]. For example, Yang *et al.* [55] proposed a multi-modal transformer-based approach for smart contract summarization, which learns source code representation from two heterogeneous modalities of the AST, *i.e.*, structure-based traversal sequences and graphs. Mi *et al.* [34] utilized feature vectors generated from byte code of smart contracts as the input of a metric learning-based deep neural network to detect vulnerabilities in smart contracts. Lutz *et al.* [33] leveraged a multi-output neural network architecture to learn specific vulnerability types from the input smart contracts. Zhou *et al.* [59] proposed an approach to generating practical inputs for testing smart contracts by using a representation vector learning model. Shi *et al.* [45] presented an MM-SCS model for semantic search of smart contract code, which captures the data flow and control flow information of the code from a contract element dependency graph.

The aforementioned works learn code representations in fully supervised settings with deep neural networks. They are designed for specific tasks. By contrast, SRCL learns representations of smart contracts on unlabeled data by leveraging self-supervised techniques. The learned representations can be generalized for various software engineering tasks.

8 CONCLUSION

We have proposed SRCL, a self-supervised representation learning approach for smart contracts. SRCL learns local and global information from the pairs of type and value sequences of smart contracts' ASTs by a Transformer and CNN encoder, then leverages three well-designed learning tasks to optimize the encoder for generating high-quality representation. Experimental results show that SRCL outperforms the state-of-the-art methods by a significant margin.

In future, we will adopt the proposed SRCL on larger-scale datasets in different programming languages and for a variety of software engineering tasks such as code-to-code search and contract name generation.

ACKNOWLEDGMENTS

This research is supported by National Natural Science Foundation of China (Grant No. 62102244, 62032004).

REFERENCES

[1] Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. 2019. code2seq: Generating Sequences from Structured Representations of Code. In *7th International Conference on Learning Representations (ICLR)*.

[2] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2019. code2vec: Learning distributed representations of code. *Proceedings of the ACM on Programming Languages.* 3, POPL (2019), 1–29.

[3] 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 (BSCI).* 47–59.

[4] Arshdeep Bahga and Vijay K Madisetti. 2016. Blockchain platform for industrial internet of things. *Journal of Software Engineering and Applications* 9, 10 (2016), 533–546.

[5] Piotr Bojanowski, Edouard Grave, Armand Joulin, and Tomás Mikolov. 2017. Enriching Word Vectors with Subword Information. *Trans. Assoc. Comput. Linguistics.* 5 (2017), 135–146.

[6] Nghi D. Q. Bui, Yijun Yu, and Lingxiao Jiang. 2021. InferCode: Self-Supervised Learning of Code Representations by Predicting Subtrees. In *43rd IEEE/ACM International Conference on Software Engineering (ICSE).* 1186–1197.

[7] Hengyi Cai, Hongshen Chen, Yonghao Song, Cheng Zhang, Xiaofang Zhao, and Dawei Yin. 2020. Data Manipulation: Towards Effective Instance Learning for Neural Dialogue Generation via Learning to Augment and Reweight. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics (ACL).* 6334–6343.

[8] Xiangping Chen, Peiyong Liao, Yixin Zhang, Yuan Huang, and Zibin Zheng. 2021. Understanding Code Reuse in Smart Contracts. In *28th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER).* 470–479.

[9] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of the Conference of the North American Chapter of the Association for*

Computational Linguistics: Human Language Technologies (NAACL-HLT). 4171–4186.
[10] Ming Ding, Jie Tang, and Jie Zhang. 2018. Semi-supervised Learning on Graphs with Generative Adversarial Nets. In *Proceedings of the 27th ACM International Conference on Information and Knowledge Management (CIKM)*. 913–922.
[11] Steven Y. Feng, Varun Gangal, Jason Wei, Sarath Chandar, Soroush Vosoughi, Teruko Mitamura, and Eduard Hovy. 2021. A Survey of Data Augmentation Approaches for NLP. In *Findings of the Association for Computational Linguistics (ACL-IJCNLP)*. 968–988.
[12] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In *Findings of the Association for Computational Linguistics (EMNLP)*. 1536–1547.
[13] Kunihiko Fukushima. 1988. Neocognitron: A hierarchical neural network capable of visual pattern recognition. *Neural Networks*. 1, 2 (1988), 119–130.
[14] Zhipeng Gao, Lingxiao Jiang, Xin Xia, David Lo, and John Grundy. 2021. Checking Smart Contracts With Structural Code Embedding. *IEEE Transactions on Software Engineering*. 47, 12 (2021), 2874–2891.
[15] Xiaodong Gu, Hongyu Zhang, and Sunghun Kim. 2018. Deep code search. In *Proceedings of the 40th International Conference on Software Engineering (ICSE)*. ACM, 933–944.
[16] Hongyu Guo, Yongyi Mao, and Richong Zhang. 2019. Augmenting Data with Mixup for Sentence Classification: An Empirical Study. *CoRR* abs/1905.08941 (2019).
[17] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 770–778.
[18] Ningyu He, Lei Wu, Haoyu Wang, Yao Guo, and Xuxian Jiang. 2020. Characterizing Code Clones in the Ethereum Smart Contract Ecosystem. In *24th International Conference on Financial Cryptography and Data Security*, Vol. 12059. Springer, 654–675.
[19] Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. 2018. Deep code comment generation. In *Proceedings of the 26th Conference on Program Comprehension (ICPC)*. ACM, 200–210.
[20] Xuan Huo, Ming Li, and Zhi-Hua Zhou. 2016. Learning Unified Features from Natural and Programming Languages for Locating Buggy Source Code. In *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence, (IJCAI)*. 1606–1612.
[21] Paras Jain, Ajay Jain, Tianjun Zhang, Pieter Abbeel, Joseph Gonzalez, and Ion Stoica. 2021. Contrastive Code Representation Learning. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP)*. 5954–5971.
[22] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stéphane Glondu. 2007. DECKARD: Scalable and Accurate Tree-Based Detection of Code Clones. In *29th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, 96–105.
[23] Tapas Kanungo, David M Mount, Nathan S Netanyahu, Christine D Piatko, Ruth Silverman, and Angela Y Wu. 2002. An efficient k-means clustering algorithm: Analysis and implementation. *IEEE transactions on pattern analysis and machine intelligence*. 24, 7 (2002), 881–892.
[24] Dongsun Kim, Yida Tao, Sunghun Kim, and Andreas Zeller. 2013. Where should we fix this bug? a two-phase recommendation model. *IEEE transactions on software Engineering*. 39, 11 (2013), 1597–1610.
[25] Seohyun Kim, Jinman Zhao, Yuchi Tian, and Satish Chandra. 2021. Code Prediction by Feeding Trees to Transformers. In *43rd IEEE/ACM International Conference on Software Engineering (ICSE)*. 150–162.
[26] Diederik P. Kingma and Prafulla Dhariwal. 2018. Glow: Generative Flow with Invertible 1x1 Convolutions. In *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems (NeurIPS)*. 10236–10245.
[27] Masanari Kondo, Gustavo Ansaldi Oliva, Zhen Ming (Jack) Jiang, Ahmed E. Hassan, and Osamu Mizuno. 2020. Code cloning in smart contracts: a case study on verified contracts from the Ethereum blockchain platform. *Empir. Softw. Eng.* 25, 6 (2020), 4617–4675.
[28] Lingpeng Kong, Cyprien de Masson d'Autume, Lei Yu, Wang Ling, Zihang Dai, and Dani Yogatama. 2020. A Mutual Information Maximization Perspective of Language Representation Learning. In *8th International Conference on Learning Representations (ICLR)*.
[29] Zhenzhong Lan, Mingda Chen, Sebastian Goodman, Kevin Gimpel, Piyush Sharma, and Radu Soricut. 2020. ALBERT: A Lite BERT for Self-supervised Learning of Language Representations. In *8th International Conference on Learning Representations (ICLR)*.
[30] Wei Li, Haozhe Qin, Shuhan Yan, Beijun Shen, and Yuting Chen. 2020. Learning Code-Query Interaction for Enhancing Code Searches. In *IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 115–126.
[31] Lukas Liebel and Marco Körner. 2018. Auxiliary Tasks in Multi-task Learning. *CoRR* abs/1805.06334 (2018).
[32] Stacy K Lukins, Nicholas A Kraft, and Letha H Etzkorn. 2010. Bug localization using latent dirichlet allocation. *Information and Software Technology*. 52, 9 (2010), 972–990.
[33] Oliver Lutz, Huili Chen, Hossein Fereidooni, Christoph Sendner, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, and Farinaz Koushanfar. 2021. ESCORT: Ethereum Smart COntRacTs Vulnerability Detection using Deep Neural Network and Transfer Learning. *CoRR* abs/2103.12607 (2021).
[34] Feng Mi, Zhuoyi Wang, Chen Zhao, Jinghui Guo, Fawaz Ahmed, and Latifur Khan. 2021. VSCL: Automating Vulnerability Detection in Smart Contracts with Deep Learning. In *IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*. 1–9.
[35] Tomás Mikolov, Ilya Sutskever, Kai Chen, Gregory S. Corrado, and Jeffrey Dean. 2013. Distributed Representations of Words and Phrases and their Compositionality. In *27th Annual Conference on Neural Information Processing Systems (NIPS)*. 3111–3119.
[36] Junghyun Min, R. Thomas McCoy, Dipanjan Das, Emily Pitler, and Tal Linzen. 2020. Syntactic Data Augmentation Increases Robustness to Inference Heuristics. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, ACL 2020, Online, July 5-10, 2020*. 2339–2352.
[37] Robin Nix and Jian Zhang. 2017. Classification of Android apps and malware using deep neural networks. In *International Joint Conference on Neural Networks (IJCNN)*. IEEE, 1871–1878.
[38] Nicole Radziwill. 2018. Blockchain revolution: How the technology behind Bitcoin is changing money, business, and the world. *The Quality Management Journal*. 25, 1 (2018), 64–65.
[39] Gözde Gül Sahin and Mark Steedman. 2018. Data Augmentation via Dependency Tree Morphing for Low-Resource Languages. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*. 5004–5009.
[40] Hitesh Sajnani, Vaibhav Saini, Jeffrey Svajlenko, Chanchal K. Roy, and Cristina V. Lopes. 2016. SourcererCC: scaling code clone detection to big-code. In *Proceedings of the 38th International Conference on Software Engineering (ICSE)*. ACM, 1157–1168.
[41] Jorge M. Santos and Mark J. Embrechts. 2009. On the Use of the Adjusted Rand Index as a Metric for Evaluating Supervised Classification. In *19th International Conference on Artificial Neural Networks (ICANN)*. Springer, 175–184.
[42] Alexander Savelyev. 2017. Contract law 2.0:'Smart'contracts as the beginning of the end of classic contract law. *Information & communications technology law*. 26, 2 (2017), 116–134.
[43] Scale. 2022. Data Pricing. [EB/OL]. https://scale.com/pricing, Accessed March 10, 2022.
[44] Rico Sennrich, Barry Haddow, and Alexandra Birch. 2016. Improving Neural Machine Translation Models with Monolingual Data. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (ACL)*.
[45] Chaochen Shi, Yong Xiang, Jiangshan Yu, and Longxiang Gao. 2021. Semantic Code Search for Smart Contracts. *CoRR* abs/2111.14139 (2021).
[46] Melanie Swan. 2015. *Blockchain: Blueprint for a new economy*. O'Reilly Media, Inc.
[47] Nick Szabo. 1997. Formalizing and Securing Relationships on Public Networks. *First Monday*. 2, 9 (1997).
[48] Laurens Van der Maaten and Geoffrey Hinton. 2008. Visualizing data using t-SNE. *Journal of machine learning research*. 9, 11 (2008), 2579–2605.
[49] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. In *Annual Conference on Neural Information Processing Systems*. 5998–6008.
[50] Hongwei Wang, Jia Wang, Jialin Wang, Miao Zhao, Weinan Zhang, Fuzheng Zhang, Xing Xie, and Minyi Guo. 2018. GraphGAN: Graph Representation Learning With Generative Adversarial Nets. In *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, (AAAI), the 30th innovative Applications of Artificial Intelligence (IAAI), and the 8th AAAI Symposium on Educational Advances in Artificial Intelligence (EAAI)*. 2508–2515.
[51] William Yang Wang and Diyi Yang. 2015. That's So Annoying!!!: A Lexical and Frame-Semantic Embedding Based Data Augmentation Approach to Automatic Categorization of Annoying Behaviors using #petpeeve Tweets. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP)*. 2557–2563.
[52] Jason W. Wei and Kai Zou. 2019. EDA: Easy Data Augmentation Techniques for Boosting Performance on Text Classification Tasks. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*. 6381–6387.
[53] Wenhan Xiong, Jingfei Du, William Yang Wang, and Veselin Stoyanov. 2020. Pretrained Encyclopedia: Weakly Supervised Knowledge-Pretrained Language Model. In *8th International Conference on Learning Representations (ICLR)*.
[54] Zhilin Yang, Zihang Dai, Yiming Yang, Jaime G. Carbonell, Ruslan Salakhutdinov, and Quoc V. Le. 2019. XLNet: Generalized Autoregressive Pretraining for Language Understanding. In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems (NeurIPS)*. 5754–5764.

[55] Zhen Yang, Jacky Keung, Xiao Yu, Xiaodong Gu, Zhengyuan Wei, Xiaoxue Ma, and Miao Zhang. 2021. A Multi-Modal Transformer-based Code Summarization Approach for Smart Contracts. In *29th IEEE/ACM International Conference on Program Comprehension (ICPC)*. 1–12.

[56] Hongyi Zhang, Moustapha Cissé, Yann N. Dauphin, and David Lopez-Paz. 2018. mixup: Beyond Empirical Risk Minimization. In *6th International Conference on Learning Representations (ICLR)*.

[57] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, Kaixuan Wang, and Xudong Liu. 2019. A novel neural source code representation based on abstract syntax tree. In *Proceedings of the 41st International Conference on Software Engineering (ICSE)*. IEEE / ACM, 783–794.

[58] Shufan Zhou, Beijun Shen, and Hao Zhong. [n.d.]. Lancer: Your Code Tell Me What You Need. In *34th IEEE/ACM International Conference on Automated Software Engineering (ASE*.

[59] Teng Zhou, Kui Liu, Li Li, Zhe Liu, Jacques Klein, and Tegawendé F Bissyandé. 2021. SmartGift: Learning to Generate Practical Inputs for Testing Smart Contracts. In *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 23–34.